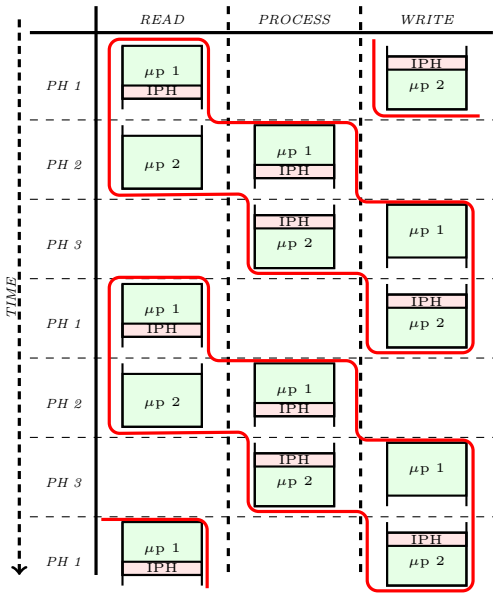
Accelerating Unstructured Mesh Computations using a Custom Streaming Architecture

Kyrylo Tkachov

Supervisor: Prof. Paul Kelly

Second marker: Dr. Tony Field



2012, Department of Computing

Imperial College London

Abstract

In this report we present a methodology for accelerating computations performed on unstructured meshes in the course of a finite volume approach. We implement a custom streaming datapath using Field Programmable Gate Arrays, or FPGAs to perform the bulk of the floating point operations required by an application. In particular, we focus on dealing with irregular memory access patterns that are a consequence of using an unstructured mesh in such a way as to facilitate a streaming model of computation. We describe the partitioning of the mesh and the techniques used to exchange information between neighbouring partitions, using so-called halos. We provide hardware accelerated version of a concrete application and develop a performance model used to predict performance and justify our design decisions and guide the implementation.

Acknowledgements

I would like to thank Professor Paul Kelly for giving me so much of his time and ideas and making me aware of scope of the project and the intricacies involved. I would also like to thank Dr. Carlo Bertolli for providing practical advice and explaining the labyrinth that is heterogenous computing. Special thanks go to the team at Maxeler Technologies for helping me out with the details of FPGA-based acceleration and providing support for their excellent toolchain. I extend my gratitude to Dr. Tony Field, my personal tutor, who supported me throughout my years at Imperial College and guided so much of my academic development, as well as being the second marker for this project.

I would like to thank my mother and grandfather for supporting me through university, both materially and psychologically. Last but not least, I would like to thank my coursemates and friends all over the world, with whom I've had many thought-provoking discussions on every subject imaginable and who always kept me motivated, even when I doubted myself.

Contents

Chapter 1

Introduction

This project presents a methodology for accelerating computations performed on unstructured meshes in the context of Computational Fluid Dynamics (CFD). We use Field Programmable Gate Arrays, or FPGAs, to construct a high-throughput streaming pipeline which is kept filled thanks to an appropriate data layout and partitioning scheme for the mesh. We explore the rearrangement and grouping schemes used to achieve locality of the data points. A formal performance model is constructed to predict the performance characteristics of our architecture and hence justify the design choices made. Appropriate evaluation tests are performed to evaluate the results on a sample CFD application, achieving speedup comparable with state of the art GPGPU and multi-processor solutions. In this section we present a general overview of the problem domain, the hardware platform and the contributions of this project.

1.1 The domain

Computational Fluid Dynamics, or CFD, is a branch of physics focused on numerical algorithms that simulate the movement of fluids and gases and their interactions with surfaces. These simulations are widely used by engineers to design structures and equipment that interact with fluid substances, for example airplane wings and turbines, water and oil pipelines etc.

The required calculations are usually expressed as systems of partial differential equations, the Navier-Stokes equations or the Euler equations, which are discretized using any of a number of techniques. The technique used by our sample application, Airfoil, is the finite volume method that

calculates values at discrete places in a mesh and relies on the observation that the fluxes entering a volume are equal to the fluxes leaving it. This project is not concerned with the exact mathematical formulation of these techniques, but they provide a feel for the origins of the problem domain.

1.2 The Airfoil program

The sample program we examine is called Airfoil, a 2D unstructured mesh finite volume simulation of fluid motion around an airplane wing (which has the shape of an airfoil). Airfoil was written as a representative of the class of programs that are tackled by OP2, a framework partially developed and maintained by the Software Performance Optimisation group at Imperial College to abstract the acceleration of unstructured mesh computations on a wide variety of hardware backends.

Airfoil defines an unstructured mesh through sets of nodes, edges and cells and associating them through mappings. Airfoil is written in the C language and these sets are represented at the lowest level as C-arrays. Then data is associated with these sets, such as node coordinates, temperature, pressure etc. The mesh solution is then expressed as the conceptually parallel application of computational kernels on the data associated with each element of a particular set (nodes, edges, cells). These kernels are usually floating point- intensive operations and update the datasets. The procedure is repeated through multiple iterations as desired until a steady-state solution is reached. A more detailed discussion of the unstructured mesh is presented in the Background section of this report.

1.3 FPGAs, streaming and acceleration

In this project we explore the acceleration possibilities of problems in the described domain by using Field Programmable Gate Arrays, or FPGAs. FPGAs are integrated circuits that can be reconfigured on the fly to implement in hardware any logic design. Thanks to this property they provide the development flexibility of software with the benefits of an explicit custom hardware datapath. At a high level, FPGAs can be viewed as a two-dimensional grid of logic elements that can be interconnected in any desirable way.

The FPGA acceleration approach we look at is the streaming model of computation. In a streaming approach we create a dataflow graph out of simple computational nodes that perform a specific operation on pieces of

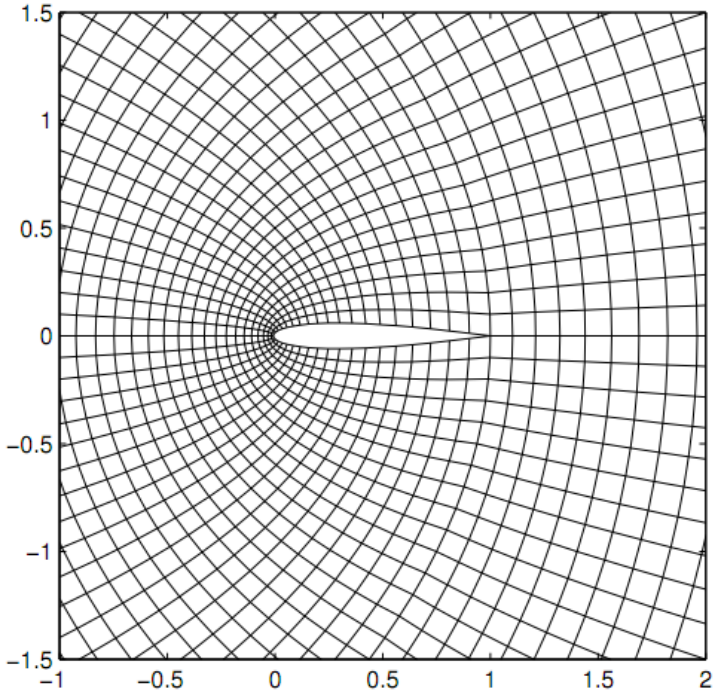


Figure 1.1: Visualisation of a reduced version of the Airfoil mesh

data pushed in and out of them. Connecting these nodes together creates a pipeline through which one can stream an array of data and get one output per cycle thus achieving high throughput. A simple dataflow graph can be seen in Figure 1.2. FPGAs are usually programmed using a low level hardware description language like VHDL or Verilog, however many tools have been designed that allow a developer to specify high-level designs. We use MaxCompiler, a compiler that lets us specify the computational graph through a high-level Java API, so we focus on the functional aspects of our design and the tool generates a hardware implementation of it. We use this approach to implement a datapath for a kernel described in Airfoil and we then look at approaches to utilise the streaming bandwidth. The FPGAs we consider have a large DRAM storage area attached (>24GB) to them that can be used to store the mesh and utilising the bandwidth of that DRAM fully is key to achieving maximum performance.

During the course of our work it emerges that in order to stream data to and from the accelerator continuously, we need to enforce some spatial locality in the mesh data, thus requiring us to reorder the data and or-

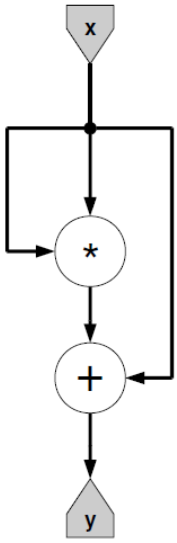


Figure 1.2: A simple dataflow graph that implements the function $y(x) = x^2 + x$. The adder and the multiplication nodes are fully pipelined, enabling a throughput of one result per cycle, thus providing us with the intuition for accelerating floating point calculations

ganise it into partitions that will be stored in the kernel internally and will need to exchange data with neighbouring partitions through a halo exchange mechanism. This opens a whole new space of decisions that we must make pertaining to the storage layout and streaming responsiblities of the DRAM and the host machine. We present the mesh partitioning schemes that are used to maximise DRAM bandwidth utilisation and maximise pipelining.

We present a performance model that will be used to describe the theoretical performance increase of the system in terms of various parameters like DRAM utilisation, clock frequency etc. Finally we evaluate the performance of our implementation of Airfoil against existing GPGPU and multi-processors cluster implementations.

1.4 Contributions

- We present an architecture for accelerating unstructured mesh computations using deeply pipelined streaming FPGA designs.
- We investigate memory layout issues that arise from efforts to max-

imise the spatial locality of the mesh and consquently increase utilisation of the streaming bandwidth. This is achieved through interleaving of I/O and computation thanks to a two-level partitioning scheme.

- We develop a scheme for handling dependencies across partitions by offloading a workload to the host machine.
- We provide a hardware accelerated version of part of the Airfoil program using the methodologies described in this report.
- We provide a predictive performance model that is used to justify our design decisions and provide a formal expression of the potential speedup.

Chapter 2

Background

This section provides more detail on the sample application, the representation of meshes, the data sets and the iteration structure of Airfoil. An overview of the Maxeler toolchain is given, which is used to implement the streaming solution we develop. The streaming model of computation is presented in the context of MaxCompiler by walking through steps to build a simple MaxCompiler application. Real number representation is discussed and the concept of a halo is introduced. Previous work in this area is presented and summarised in order to provide a context for the contributions of our work.

2.1 Unstructured meshes and their representation

The spatial domain of the problem can be discretised into either a structured or an unstructured mesh. A structured mesh has the advantage of having a highly regular structure and thus a highly predictable access pattern. If, however, one needs a more detailed solution around a particular area, the mesh would would have to be fine-grained across the whole domain, thus increasing the number of cells, nodes and edges by a large factor even in areas that are not of such great interest. This is where unstructured meshes come in. They explicitly describe the connectivity between the elements and can thus be refined and coarsened around particular areas of interest. This provides much greater flexibility at the expense of losing the regularity of the mesh, forcing us to store the connectivity information that defines its topology. It is useful to have an intimate understanding of the representation of unstructured meshes in order to understand the techniques discussed further on. A graphical example is shown in Figure 2.1

In our sample application the mesh distinguishes three main elements: nodes, cells and edges. We have to represent the connectivity information between them. This is done through *indirection maps* that store, for example, the nodes that an edge connects or the nodes that a cell contains. In the application we explore the cells always have four nodes and the edges always connect two nodes and have two cells adjacent. In the more general case of variable-dimension cells (quadrilaterals, triangles, hexagons all mixed together) we would need an additional array storing the indices into the indirection maps and the sizes of the elements. But we do not consider such meshes here.

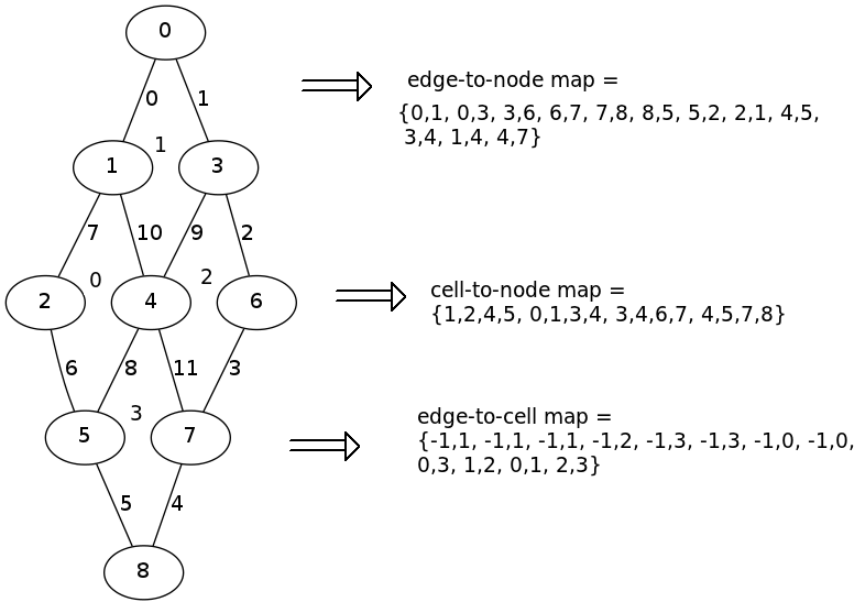


Figure 2.1: An example mesh and its representation using indirection arrays. The cell numbers are shown inside the quadrilaterals formed by the nodes (circles) and edges (edges connecting the nodes). Together with the indirection map, we also store an integer $dim \in \mathbb{N}$ which specifies the dimension of the mapping. Thus, the data associated with element i are stored in the range $[i * dim, \ldots, i * (dim + 1) - 1]$ of the relevant indirection map (in the example: the nodes associated with edge 3 are stored at indices $3 * 2 = 6$ and $3 * 2 + 1 = 7$). Note: in the edge-to-cell map -1 represents a boundary cell that may be handled in a special way by a computational kernel.

The above method deals with the connectivity information amongst the different elements of the mesh. The data on which we perform the arith-

metic calculations is stored in arrays indexed by element number. Such an approach is presented in Figure 2.2.

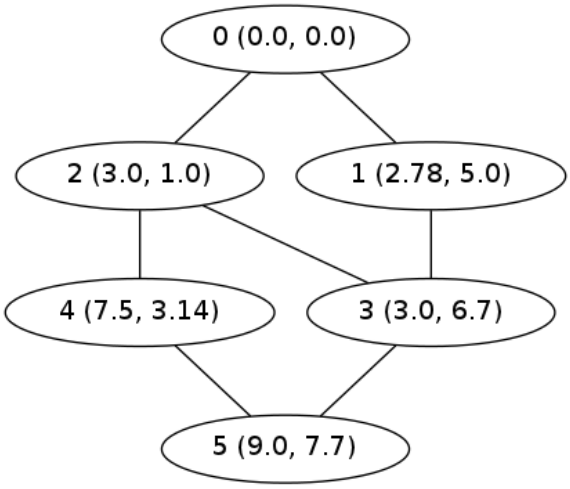


Figure 2.2: An example mesh with coordinate data associated with each node ((x, y) from *node_id* (x, y)). The coordinate data will be represented as an array of floating point numbers $x = \{0.0, 0.0, 2.78, 5.0, 3.0, 1.0, 3.0, 6.7, 7.5, 3.14, 9.0, 7.7\}$. Again we also record the dimension of the data (in this case $dim = 2$) in order to access the data set associated with each element. In this example, the coordinate data for node 4 is stored at indices $4 * 2 = 8$ and $4 * 2 + 1 = 9$ of the array x.

2.2 Airfoil

Airfoil was written as a representative of the class of problems we are interested in. It was initially designed as a non-trivial example of the issues tackled by the OP2 framework. Although we are not directly dealing with OP2 in this project, an overview of Airfoil within this context is provided by MB Giles et al [1] because it discusses the acceleration issues arising from the memory access pattern.

The computational work in Airfoil is performed by 5 loops that work one after the other and operate on the nodes, cells and edges of the mesh. Each loop iterates over the cells or edges of the mesh, applying a computational kernel to elements of a data set associated with each mesh element. Conceptually, the application of a kernel to a data item is independent of the

application to any other item in the same set, and can therefore be executed in parallel with an application of the same kernel on another mesh element. The complexity comes from reduction operations, where some edges or cells update the same data item (e.g associated with the same cell). In these cases care must be taken to ensure the correct update of the data. For parallel architectures such as GPUs and multi-processor clusters this issue can be resolved by enforcing an atomic commit scheme or by colouring the mesh partitions, so that no two partitions update the same data item simultaneously [1].

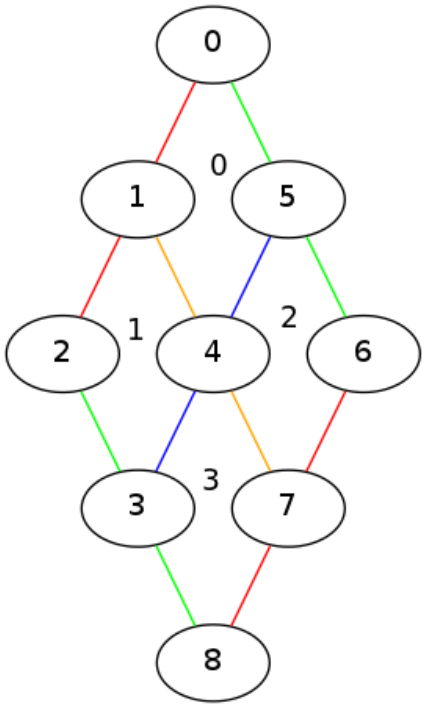


Figure 2.3: An example mesh, showing data dependencies between edges that affect cell data.

Consider Figure 2.3. Take for example edges $\alpha = (1, 4)$ and $\beta = (4, 5)$. Suppose there is a data item x associated with every cell and the processing of an edge increments the data items associated with its two cells. α and β cannot execute in parallel because they are both associated with cell 0 and can therefore end up using out of date copies of the data associated with cell 0 by the following sequence of events: α reads initial x_0, β reads

x_0, α computes $x_\alpha = x_0 + 1$, β computes $x_\beta = x_0 + 1$, α writes back x_α, β writes back x_β and the final value of x turns out to be $x_\beta = x_0 + 1$ instead of the desired $x_0 + 2$. Some implementations work around this issue by colouring the edges, such that no two edges of the same colour share a cell and can therefore be processed in parallel. Figure 2.3 shows such a colouring. Another option would be to introduce atomic operations and/or locking, but that would serialise the memory accesses, reducing parallelism, or requiring more exotic memory architectures (i.e one that reduces all of the memory write requests with addition before commiting the result).

2.2.1 Computational kernels and data sets

Airfoil defines five computational kernels that iterate over the mesh, performing floating point calculations on the data sets defined over the elements of the mesh. We shall describe them by the elements they iterate over and by the elements they read and modify. As described above, we also define some data sets that are associated with the mesh elements. The datasets defined in Airfoil are shown in Table 2.1.

Data set name	Associated with	Type/Dimension	Physical meaning
x	Nodes	$\mathbb{R} \times \mathbb{R}$	Node coordinates
q	Cells	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$	density, momentum, energy per unit volume
q_old	Cells	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$	values of q from previous iteration
res	Cells	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$	residual
adt	Cells	$\mathbb{R}$	Used for calculating area/timestep
bound	Edges	$\{0, 1\}$	Specifies whether an edge is on the boundary of the mesh

Table 2.1: Table showing the data sets and their types. In the actual implementation, we may choose to represent real numbers ($\mathbb{R}$) as standard or double precision floating point numbers or as fixed point numbers (discussed later). Elements of dimension larger than one will be represented as arrays. The physical meaning of these sets is not important, however Airfoil is generally intereseted in computing a steady-state solution for the q data set.

The kernels are presented in Table 2.2 along with the datasets they require and modify.

Kernel Name	Iterates over	Reads	Writes
save_soln	Cells	q	q_old
adt_calc	Cells	x, q	adt
res_calc	Edges	x, q, adt	res
bres_calc	(Boundary) Edges	x, q, adt, bound	res
update	Cells	q_old, adt, res	q, res

Table 2.2: Table showing the kernels defined in airfoil and their data requirements.

To show a more concrete example of what these kernels do, the res_calc kernel code in the C language is presented in Listing 1. The rest of the kernels are reproduced in A.1. The res_calc kernel applied during an iteration

over edges and requires data sets associated with the two nodes and two cells that each edge references.

```
1  void res_calc(float *x1, float *x2, float *q1, float *q2,
2                float *adt1, float *adt2, float *res1, float *res2) {
3    float dx,dy,mu, ri, p1,vol1, p2,vol2, f;
4    dx = x1[0] - x2[0];
5    dy = x1[1] - x2[1];
6    ri = 1.0f/q1[0];
7    p1 = gm1*(q1[3]-0.5f*ri*(q1[1]*q1[1]+q1[2]*q1[2]));
8    vol1 = ri*(q1[1]*dy - q1[2]*dx);
9    ri = 1.0f/q2[0];
10   p2 = gm1*(q2[3]-0.5f*ri*(q2[1]*q2[1]+q2[2]*q2[2]));
11   vol2 = ri*(q2[1]*dy - q2[2]*dx);
12   mu = 0.5f*((*adt1)+(*adt2))*eps;
13   f = 0.5f*(vol1* q1[0] + vol2* q2[0] ) + mu*(q1[0]-q2[0]);
14   res1[0] += f;
15   res2[0] -= f;
16   f = 0.5f*(vol1* q1[1] + p1*dy + vol2* q2[1] + p2*dy) + mu*(q1[1]-q2[1]);
17   res1[1] += f;
18   res2[1] -= f;
19   f = 0.5f*(vol1* q1[2] - p1*dx + vol2* q2[2] - p2*dx) + mu*(q1[2]-q2[2]);
20   res1[2] += f;
21   res2[2] -= f;
22   f = 0.5f*(vol1*(q1[3]+p1) + vol2*(q2[3]+p2) ) + mu*(q1[3]-q2[3]);
23   res1[3] += f;
24   res2[3] -= f;
25 }
```

Listing 1: Definition of the res_calc kernel with reals represented as single precision floating point numbers. Note the type signature. The kernel requires the element of the dataset x associated with each of the two nodes of the edge we are currently processing and the q, adt and res elements of the two cells associted with the current edge. Note that the res set is updated by incrementing ($+=$), introducing dependencies between parallel applications of the kernel to different edges. The important part of this definition are the data requirements of the kernel and not the exact meaning of the arithmetic operations. The variables $gm1$ and eps are global constants that do not need to be passed in explicitly.

2.2.2 Indirection maps

Having defined the data sets and the kernels, we now need to define the indirection maps that express the connectivity of the mesh and the relationships between the elements of the mesh. Airfoil has five such maps called: edge, cell, ecell, bedge, becell. They are presented in Figure 2.4.

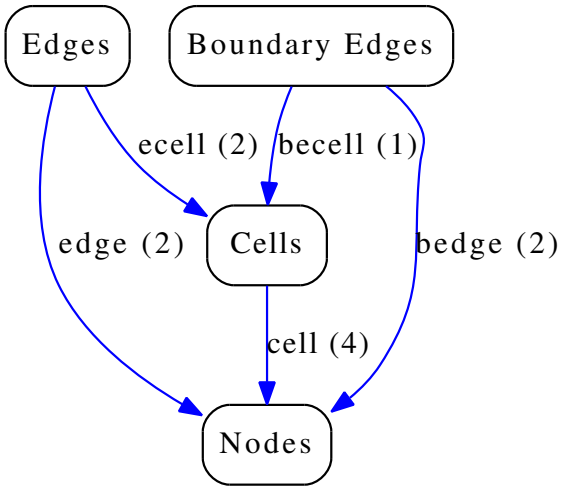


Figure 2.4: Diagram showing the maps between the mesh elements The dimension of the map is shown in parentheses next to the name. Thus the map *edge* relating edges to nodes with dimension 2 means that for each edge, there are two nodes associated with it.

Having specified the data sets, indirection maps and kernels, the application of a kernel on an element is performed by looking up the mesh elements that element is associated with through the indirection maps and using those to access the data sets required by the kernel. For example, the invokation of the res_calc kernel defined in Listing 1 can be done with the following line of C:

```
res_calc(
        &x[2*edge[2*i]], &x[2*edge[2*i+1]], &q[4*ecell[2*i]],
        &q[4*ecell[2*i+1]], &adt[ecell[2*i]], &adt[ecell[2*i+1]],
        &res[4*ecell[2*i]], &res[4*ecell[2*i+1]]
        );
```

Recall that res_calc operates on edges. There is a double level of indirection going on here. `i` is the number of the edge we are currently processing (`i`

ranges in $[0..number\_of\_edges - 1]$). As described in the section on mesh representation, the two nodes corresponding to the edge are stored at indices `2*i` and `2*i+1` of the *edge* map. For each of those nodes, res_calc requires the corresponding element in the x data set. Recall from Table 2.1 that the x set has a dimension of 2. Therefore the node numbers acquired from `edge[2*i]` and `edge[2*i+1]` are multiplied by 2 and used as indices into the array x to access the correct data. Similarly for the rest of the arguments.

The complete iteration step in a sequential implementation of Airfoil is shown in Listing 2. The old values of q are stored in q_old and the inner loop runs twice before saving the solution again. The metric rms is computed in each iteration that is used to measure the convergence of the solution. The variables *ncell*, *nedge*, *nbedge* represent the number of cells, the number of edges and the number of boundary edges respectively.

A run of the sequential version in Listing 2 on a mesh with 721801 nodes, 1438600 edges, 2800 boundary edges and 720000 cells on a machine with an Intel Core i7-2600 CPU at 3.4 GHz takes about 115.6 seconds to complete 2000 iterations. The time spent in each kernel is presented in Table 3.1. It is evident that the computation is dominated by the res_calc and adt_calc kernels.

In this project we will be concentrating on accelerating the res_calc kernel because it is the most computationally intensive kernel and because it has the most complex data access patterns that make it the interesting case to study. Finding a way to accelerate res_calc will pave the way for accelerating any similar kernel.

Kernel Name	Time spent (seconds)	Percentage of total time (%)
save_soln	1.84	1.59
adt_calc	51.09	44.19
res_calc	53.99	46.70
bres_calc	0.23	0.20
update	8.44	7.30

Table 2.3: Table showing the time spent in each kernel during a run of a single-threaded sequential version of Airfoil on a current CPU. The total run time is 115.6 seconds.

```
int niter = 1000;
float rms = 0.0;
for(int iter=1; iter<=niter; iter++) {
  for (int i = 0; i < ncell; ++i) {
    save_soln(&q[4*i], &qold[4*i]);
  }
  for(int k=0; k<2; k++) {

    for (int i = 0; i < ncell; ++i) {
      adt_calc(&x[2*cell[4*i]], &x[2*cell[4*i+1]],
               &x[2*cell[4*i+2]], &x[2*cell[4*i+3]],
               &q[4*i], &adt[i]
              );
    }

    for (int i = 0; i < nedge; ++i) {
      res_calc(&x[2*edge[2*i]], &x[2*edge[2*i+1]],
               &q[4*ecell[2*i]], &q[4*ecell[2*i+1]],
               &adt[ecell[2*i]], &adt[ecell[2*i+1]],
               &res[4*ecell[2*i]], &res[4*ecell[2*i+1]]
              );
    }

    for (int i = 0; i < nbedge; ++i) {
      bres_calc(&x[2*bedge[2*i]], &x[2*bedge[2*i+1]],
                &q[4*becell[i]], &adt[becell[i]],
                &res[4*becell[i]], &bound[i]
               );
    }

    rms = 0.0;
    for (int i = 0; i < ncell; ++i) {
      update(&qold[4*i], &q[4*i], &res[4*i], &adt[i], &rms);
    }
  }
  rms = sqrt(rms/(float) ncell);
  if (iter%100 == 0)
    printf(" %d %10.5e \n",iter,rms);
}
```

Listing 2: The iteration structure of Airfoil.

2.3 Hardware platform, Maxeler toolchain and the streaming model of computation

The toolchain we use for implementing the FPGA accelerator is the one developed and maintained by Maxeler Technologies. It consists of the MAX3 cards that contain a Xilinx Virtex-6 chip [4] and up to 48GB of DDR3 DRAM. These cards can be programmed through MaxCompiler[5], which provides a Java-compatible object-oriented API to specify the dataflow graph. MaxCompiler will then schedule the graph, i.e. it will insert buffers that will introduce the appropriate delays in the design that will ensure the correct values will reach the appropriate stages in the pipeline at the correct clock cycle. After MaxCompiler has generated the VHDL code control is given to the Xilinx ISE tools[3] that will synthesize, place and route the design and generate a bitstring that can be used to configure the FPGA.

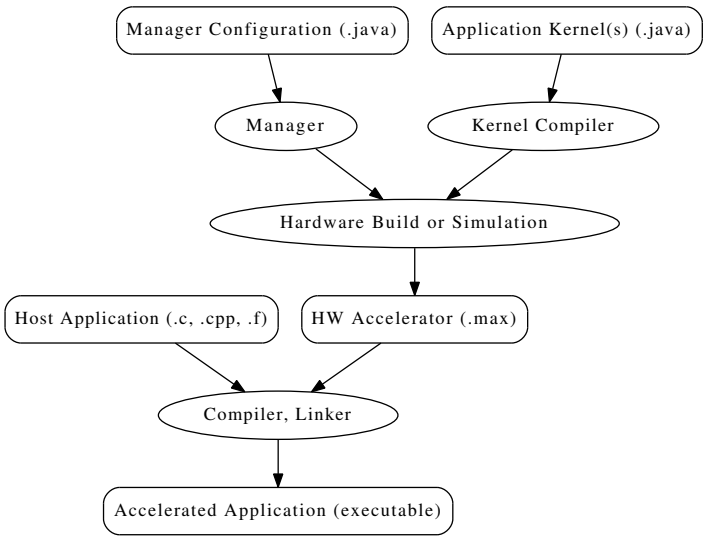


Figure 2.5: A diagram of the Maxeler toolchain. The data-flow graphs of the computational kernels are defined using a Java API. A manager connects multiple kernels together and handles the streaming to and from the kernels of data. These are combined by MaxCompiler and compiled into a .max file that can then be linked to a host C/C++ or Fortran application using standard tools (gcc, ld etc).

It will then produce a hardware design in VHDL that will then be further be compiled down to a binary bitstream that configures the FPGA

by the Xilinx proprietary tools. The bitstream is then included in what is termed a *maxfile* that contains various other meta-data about the design such as I/O stream names, named memory and register names, various run-time parameters etc. The maxfile can be linked against a normal C/C++ application using standard tools (gcc, ld etc). The interaction with the FPGA is performed by a low-level runtime: MaxCompilerRT and a driver layer: MaxelerOS. A diagram of the toolchain is shown in Figure 2.5 [5].

Computational kernels in MaxCompiler have input streams that are pushed through a pipelined dataflow graph and some of them are output from the kernel. Programmatically, a hardware stream is seen as analogous to a variable in conventional programming languages. It's value potentially changes each cycle.

2.3.1 MaxCompiler example

Listing 3 shows a MaxCompiler design that computes a running 3-point average of a stream of floating point values (32 bits).

```
1  pulic class MovingAverageKernel extends Kernel {
2
3    public MovingAverageKernel(KernelParameters parameters) {
4      super(parameters);
5      HWType flt = hwFloat(8,24);
6      HWVar x = io.input("x", flt ) ;
7      HWVar x_prev = stream.offset(x, -1);
8      HWVar x_next = stream.offset(x, +1);
9      HWVar cnt = control.count.simpleCounter(32, N);
10     HWVar sel_nl = cnt > 0;
11     HWVar sel_nh = cnt < (N-1);
12     HWVar sel_m = sel_nl & sel_nh;
13     HWVar prev = sel_nl ? x_prev : 0;
14     HWVar next = sel_nh ? x_next : 0;
15     HWVar divisor = sel_m ? 3.0 : 2.0;
16     HWVar y = (prev+x+next)/divisor;
17     io.output("y" , y,  flt);
18   }
19 }
```

Listing 3: A MaxCompiler definition of a kernel that computes a moving 3-point average with boundary conditions. Note that the arithmetic operators as well as the ternary if operator have been overloaded for HWVar objects that represent the value of a hardware stream.

MaxCompiler code is written in a Java-like language called MaxJ that provides overloaded operators such as $+, -, *, /$ and ? : . The example in Listing 3 creates a computational kernel that computes a stream of running 3-point averages,named y, from a stream of input values x. The HWVar class is the main representation of the value of a hardware stream at any clock cycle. HWVars always have a HWType that expresses the type of the stream (i.e. an integer, a floating point number, a 1-bit boolean value etc). The $stream.offset(x, -1)$ and $stream.offset(x, +1)$ expressions on lines 7 and 8 extract HWVars for the values of the stream on cycle in the past and one cycle in the future (note that this is internally done by creating implicit buffers, or FIFOs, and scheduling the pipeling accordingly). The ternary if operator ? : creates multiplexers in hardware that express choice. A Java API is provided that contains various useful design elements, such as counters (HWVars that increment their values in many configurable ways

every cycle) that can be accessed through the control.count field.

The resulting dataflow graph can be seen in Figure 2.6

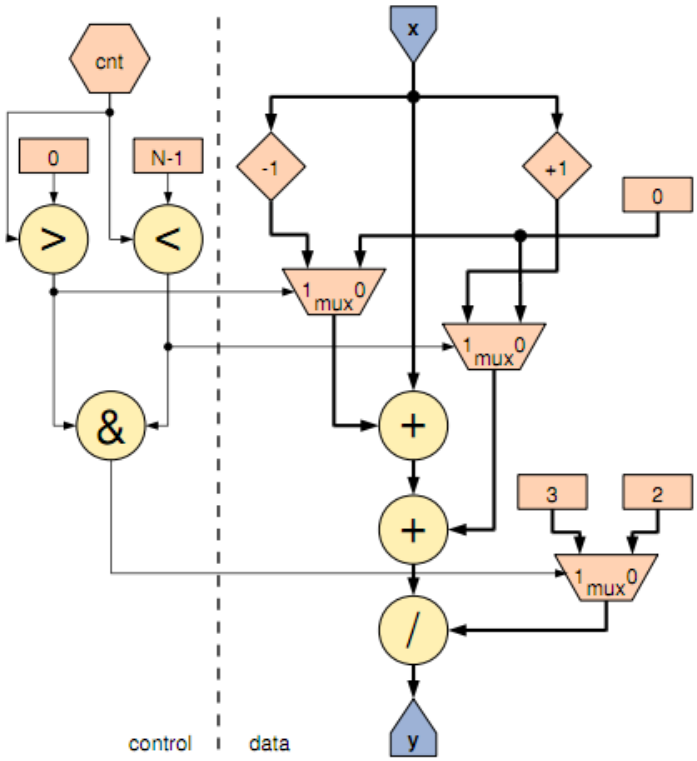


Figure 2.6: The dataflow graph resulting from the code in Listing 3. Note: the $stream.offset(x, -1)$ and $stream.offset(x, +1)$ expressions are shown here using the rhombuses with $+1$ and -1, evaluating the value of x one cycle in the 'future' and one cycle in the past respectively. The other nodes have the obvious meanings.

Kernel designs form part of a MaxCompiler design. The user also specifies a manager that describes the streaming connections between the kernels. A manager can be used to configure a design to stream data to and from the host through PCIe or from the DRAM that is attached to the FPGA. In the manager design, the user will instantiate the kernels and connect them up. Thus for the example in Listing 3 the manager might look like the one in Listing 4.

```
pulic class MovingAvgManager extends CustomManager {

  public MovingAvgManager(MAXBoardModel board_model,
                          boolean is_simulation, String name) {
    super(is_simulation, board_model, name);
    KernelBlock k
      = addKernel(
          new MovingAverageKernel(makeKernelParameters("MovingAverageKernel"))
                );

    Stream x = addStreamFromHost("x");
    k.getInput("x") <== x;

    Stream y = addStreamToHost("y");
    y <== k.getOutput("y");
  }
}
```

Listing 4: Manager specification for a MovingAverageKernel that streams the input data "x" from the host and streams the output data "y" to the host. The <== operator means connect the right hand side stream to the left hand side stream. The above code instantiates the MovingAverageKernel, creates a stream called "x" from the host and connects it to the input stream "x" in the kernel. Then it creates a stream to the host called "y" and connects to it the output stream "y" from the kernel.

After we have specified a manger, we can build the design in order to create the .max file using the following lines of code:

```
public class MovingAvgHWBuilder {
  public static void main(String argv[]) {

    MovingAvgManager m
      = new MovingAvgManager(MAX3BoardModel.MAX3242A,
                             false,
                             "MovingAverage");
    m.build() ;
  }
}
```

This builds our design for a MAX3 card (containing a Xilinx Virtex6 FPGA) using the "MovingAverage" name for the design. The second argument to the constructor (`false`) signifies that we are building a hardware design and not a simulation.

Now that we have a .max file, we can interact with the FPGA from the host code by using the MaxCompilerRT API, an example of which is shown in Listing 5. In order to use the FPGA we must initialise the maxfile as in line 14 and open the device (line 15). The actual streaming to and from the FPGA is done using the max_run vararg function (line 22) where the arrays corresponding to the input data and the allocated space for the output data are specified. The MaxCompilerRT runtime and the MaxelerOS drivers handle the low-level details of PCIe streaming and interrupts.

```
#include<stdlib.h>
#include<stdint.h>
#include<MaxCompilerRT.h>
#define DATA_SIZE 1024

int main(int argc, char* argv[]) {
        char* device_name = "/dev/maxeler0";
        max_maxfile_t* maxfile;
        max_device_handle_t* device;
        float *data_in, *data_out;

        maxfile = max_maxfile_init_MovingAverage();
        device = max_open_device(maxfile, device_name);

        data_in = (float*)malloc(DATA_SIZE * sizeof(float));
        data_out = (float*)malloc(DATA_SIZE * sizeof(float));

        for (int i = 0; i < DATA_SIZE; ++i) {
                data_in[i] = i;
        }

        max_run(device,
                max_input("x", data_in, DATA_SIZE * sizeof(float)),
                max_output("y", data_out, DATA_SIZE * sizeof(float)),
                max_runfor("MovingAverageKernel", DATA_SIZE),
                max_end());


        for (int i = 0; i < DATA_SIZE; ++i) {
                printf("data_out@%d = %f\n", i, data_out[i]);
        }

        max_close_device(device);
        max_destroy(maxfile);
        return 0;

}
```

Listing 5: A sample host code using the MaxCompilerRT API for the C language.

2.3.2 Hardware

The MAX3 card we use provides 48GB of DRAM that can be accessed with a maximum bandwidth of 38GB/s and a PCIe connection to the host machine that achieves a maximum bandwidth of 2GB/s in both directions. The Virtex6[4] FPGA by Xilinx used in the MAX3 card has about 4MB of fast on-board block RAM that should not be confused with the external DRAM. The host machine can communicate with the card through the PCIe bus using the MaxCompilerRT API. The external DRAM will be used to store the bulk of the mesh data and therefore achieving maximum utilisation of it is be one of the focal points of this project. The top-level parts of the hardware we are dealing with are presented in Figure 2.7.

The FPGA provides a number of resources that can be used to specify a design. The Xilinx Virtex6 chip we are using defines four such elements:

- LUTs: LookUp Tables are small elements of combinatorial logic that can be configured to implement any logical function.
- Flip Flops: Stateful elements that can be used as registers to implement accumulators, pipeline stages etc.
- BRAMs: Block RAMs are memory cells that are on the chip itself and can be accessed with very low latency.
- DSPs: Elements custom tuned for fast multiplication.

When building an FPGA design, one must be careful to not use more resources than the chip has to offer, therefore these numbers place a limit on the partition size we can store on the chip at any time, the number of arithmetic pipelines available etc.

The card is connected to the host machine via a PCIe bus, a popular standard that is found on most modern motherboards.

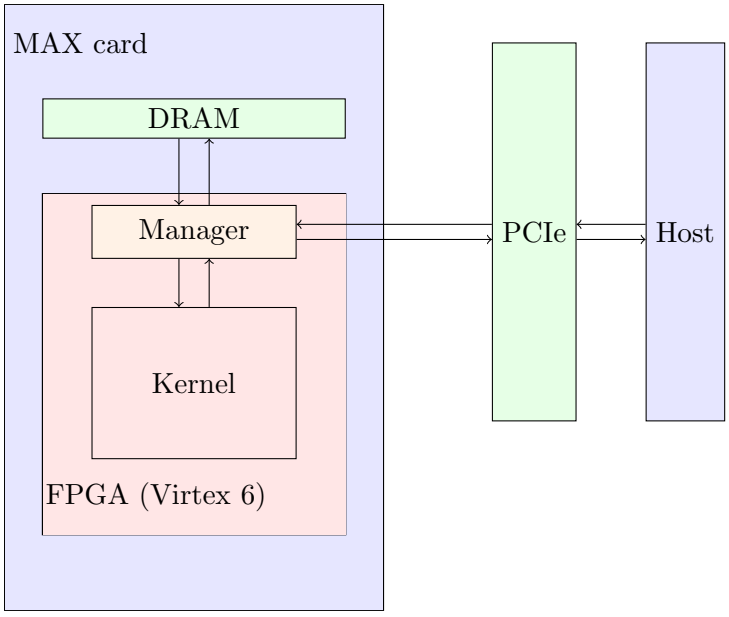


Figure 2.7: Diagram of the hardware parts of a MAX card, showing the relationships between the DRAM, PCIe, the host and the FPGA.

2.3.3 Mesh partitioning and halos

In computing the optimal memory layout for our application, we have to partition large meshes into partitions that fit in the block RAM of the FPGA. We use a popular and widely available set of tools called METIS developed by George Karypis [6] that uses state of the art techniques to partition meshes, graphs, hypergraphs and matrices according to various parameters like size, edge/hyperedge cut, minimising certain metrics etc. It is a highly robust and efficient tool that we use through its C API. Since an iteration over a mesh element may require data associated with another element, partitions have a set of elements called a *halo region* that consists of all the elements (cells, nodes, edges) that maybe accessed from another partition. Consider Figure 2.8. The partitioning is shown with the red line. The four partitions share cells (shown in purple), edges (shown in red) and nodes (along the red line). This presents a difficulty when computing an edge that uses cells in the halo region of another partition, since we are storing a single partition at a time on the device. The method used to deal with this issue is called the *halo exchange mechanism* and it opens up a large design space, with decisions usually dependent on the hardware and

communication frameworks.

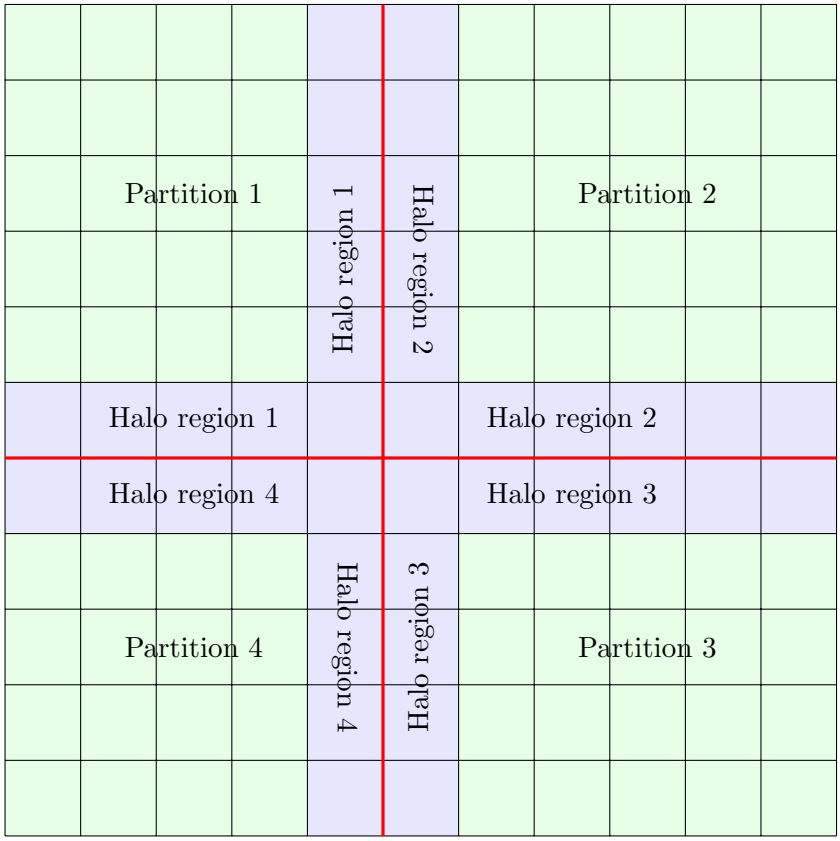


Figure 2.8: A mesh partitioned into 4 partitions, shown in green. The halo regions are shown in purple. Nodes, cells and edges belonging to the halo region can be accessed by another partition.

2.3.4 Floating point vs fixed point arithmetic

As presented in Table 2.1, the most important data sets in Airfoil (q, adt, res) consist of real numbers. Therefore a decision must be made on the low level number representation. The most common representation for real numbers in most modern architectures is the IEEE-754[7] floating point representation. This representation stores the number using three fields: the sign bit, the mantissa and the exponent. The representation of a 32-bit floating point number is shown in Figure 2.9. The standard also specifies double precision floating point numbers using 64 bits: 11 bits for the exponent and

53 bits for the mantissa. In general, a floating point number with N bits for the exponent can represent a range from $\pm 1 \times 2^{-\frac{2^N}{2}-2}$ to $\pm 2 \times 2^{\frac{2^N}{2}-1}$. For 32-bit single precision numbers, this range is $[\pm 1 \times 2^{-126} .. \pm 2 \times 2^{127}]$. The details of how the mantissa, the exponent and the sign bit are used to encode a real number are presented in the IEEE specification [7]. The decoding of a floating point number involves multiplying the mantissa by 2 raised to the power of the exponent, which is typically an expensive operation. Further details, like exponent bias and normalisation are not discussed here as they are tangential to this section.

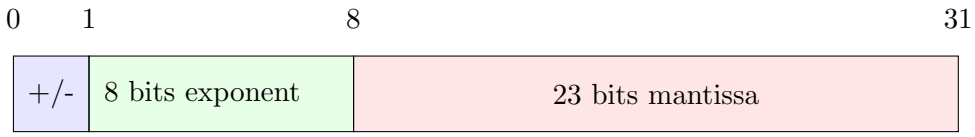


Figure 2.9: The representation of an IEEE-754 single precision floating point number. It has 8 bits for the exponent and 24 bits for the mantissa. However, one bit of the mantissa is used to represent the sign, and is therefore unavailable to the rest of the mantissa.

Since we are working with custom hardware, we have an alternative to floating point numbers for our representation of real arithmetic. We can store a real number as an integer and a fractional part at fixed offsets. This approach makes the arithmetic much simpler and hence faster, but it sacrifices range and accuracy. Fixed point representation is used in applications where the range of the numbers is predictable and not too large, or when the inputs are of limited precision. Faster in this context means fewer cycles taken to perform an operation, which translates to pipeline stages. For example, a fixed point number with 8 integer bits and 8 fraction bits using two's complement mode for negative numbers can repressent numbers from -128 up to $127 + 255/256$ in increments of $1/256$ with each fraction bit representing $1/256$.

2.4 Previous work

Computation using unstructured meshes is widely used in many areas of engineering, not just in fluid dynamics, and there have been many attempts to augument the computation using accelerators. A recent trend has been to use the many cores available on Graphics Processing Units (GPUs) to launch thousands of threads in parallel, exploiting the parallel nature of many of

these problems. Other hardware platforms include many-core architectures [1] and large parallel clusters [2]. A project close to this one is OP2 [1]. It is a framework used to specify computations on unstructured meshes in a hardware-agnostic way, allowing the user to concentrate on the functional specification of the algorithm. Airfoil is one of the test programs for that project.

More relevant to this project, there have been attempts to use FPGAs to accelerate such computations. The principles of acceleration using FPGAs are quite different compared to using GPUs or many-core systems. In the case of FPGA acceleration, the performance advantage comes from a custom, application specific, deeply pipelined datapath, often thousands of cycles deep that provides a throughput of one result per cycle. This argument for FPGA acceleration is widely accepted and is the source of the speedups achieved in all current attempts. Given this deep custop pipeline, it is a challenge for the developer to keep the pipeline fully utilised for the maximum amount of time and the techniques used to achieve this have been the main differentiating factors in the applications existing today. The most common approach has been to use on-chip block RAM memory to cache small parts of the mesh and operate on it, cache the results and write them back to main memory.

M.T. Jones and K.Ramachandran [9] formulate the unstructured mesh computation as a sparse matrix problem, $Ax = y$ where A is a large sparse matrix representing the mesh and x is the vector that is being approximated. Their approach uses the conjugate gradient method to iteratively refine the approximation of the x vector. This involves, most importantly, a multiplication of the sparse matrix A with the vector x which forms the bulk of the computation and a subsequent refinement of the mesh and reconstruction of the spares matrix. They formulate the problem as a sparse matrix-vector multiplication, whereas we are interested in iterating computational kernels over the mesh. Furthermore, they are concerned with mesh adaptation and the reconstruction of the sparse matrix in each iteration. We assume a static mesh specified at the beginning of the program.

Morishita et al. [10] examine the acceleration of CFD applications and in particular the use of on-chip block RAM resources to buffer the data in order to keep the arithmetic pipeline as full as possible. This is a more similar approach. However, their approach applies a constant stencil to a grid in 3D and tries to cache points in the grid that will be accessed in the next iteration, thus elimintating redundant accesses to the external memory. This caching/buffering is made possible by the fact that the stencil is of constant shape and thus the memory accesses can be predicted. In our application

we have a 2D mesh that does not exhibit this property.

Sanchez-Roman et al. [11] present the acceleration of an airfoil-like unstructured mesh computation using FPGAs. Their solution uses two FPGAs on a single chip that perform different calculations and they identify the need to reason about computation and data access separately. They mention the need to partition larger meshes but they do not discuss techniques for partitioning or the issues arising from data dependencies across partitions. They mention the degradation in performance arising from the unstructured memory access patterns causing cache misses. We present a technique to reorganise the mesh so as to facilitate more well-behaved memory accesses, allowing us to stream data to the datapath efficiently.

In another attempt, Sanchez-Roman et al. [12] recognise the update dependency that occurs during reduction operations, similar to the ones in our res_calc kernel and work around it by adding an accumulator that correctly updates the required data sets. However, their design is used on comparatively small meshes of a maximum of 8000 nodes. Thus all the data can fit into the on-board memory of the FPGA, eliminating the need to consider partitioning issues. The applications we are concerned with usually have meshes of the order of 10^6 edges, which will definitely not fit on the on-chip block RAMs any time soon, thus presenting the need for partitioning.

The existing work gives us many hints towards design decisions. While it was tempting to use fixed point arithmetic for the calculations, Sanchez-Roman et al. [11] hint at the difficulty in porting application from the host side to fixed point and also mention great difficulties that arise in the debugging and verification of the calculations. Furthermore, Durbano et al. [13] find that the error of fixed-point arithmetic accumulates with each iteration in finite difference applications such as the one in Airfoil. They explore the possibility of acceleration of electromagnetics calculations using the Finite Difference Time Domain method (FTDT) which has some similarities to ours from a computational point of view. These factors lead us to discard fixed point representation of real numbers in our solution. All authors mention the relative difficulty of developing FPGA-based systems compared to normal CPU implementations, citing the different mindset required and the inherently more low level reasoning about hardware, architecture and algorithms that must be done to extract the required performance characteristics.

All attempts recognise the need to tame the unstructured memory accesses that arise, and all of them use the on-chip block RAM resources to cache or buffer data before feeding it to the arithmetic pipeline, relying

on complex memory access address generators to handle memory accesses. Our approach will also use block RAMs to store parts of the mesh, but we will remove the need for complex memory access patterns by reordering, partitioning and laying out the mesh data in a way that facilitates large, contiguous bursts of streaming. To do that we add a mesh preprocessing stage on the host side that partitions and reorganises the layout in memory of the data.

Some exploration has been done in adapting meshes for particular memory architectures or reordering computations to optimise memory accesses. White B. et al. [14] have explored techniques for reordering computations on CPUs to facilitate efficient memory access, but not from a streaming perspective. There has been some work done in storing meshes for efficient access[15], but it has been focused on block-based CPU and GPU caches. We explore issues that are associated with laying out mesh data for optimal DRAM bandwidth utilisation on FPGAs.

The differentiating factors of this project from existing work are:

- We explore the architectural design space for the accelerator in conjunction with mesh reordering techniques custom-fitted for the chosen architecture. One is custom made for the other. Previous attempts do not go any further than doing simple partitioning.

- Our design aims to work for larger meshes, in the order of 10^5 - 10^6 nodes.

- We attempt to keep the design of the hardware accelerator as simple as possible, without complex memory access patterns and relying on host-side preprocessing to Figure out the optimal data layout. We are focused on maximising DRAM bandwidth utilisation.

Chapter 3

Design and Modelling

In this section we discuss the design of the hardware accelerated version of the res_calc kernel from Airfoil. We present the architecture for the FPGA-based accelerator and develop a formal performance model that is used to justify the viability of the architecture. From that point we decide on an optimal data layout that will lead to an implementation of the mesh pre-processing on the host.

3.1 DRAM and mesh storage

The MAX3 cards we have available have a large DRAM memory attached to them, and it is a natural candidate for storing mesh data. The computational kernel on the FPGA can access this memory at a maximum bandwidth of 38GB/s. This bandwidth, however, is only achievable when the memory is accessed in large bursts of contiguous addresses. Random access to this memory, while possible, is very inefficient and wasteful (because the DRAM controller will still read a large chunk of data, but return only the small fraction requested). Because of the representation of the unstructured mesh through indirection maps, processing an edge in res_calc requires a lookup in the *edge* and *ecell* maps, and then a lookup into the x, q and adt data sets. If we perform such a two-level dereferencing procedure on the DRAM, the performance is expected to degrade to a point where it is not worth considering. The on-chip block RAMs, on the other hand, are designed to be accessed randomly and do not degrade in performance when accessed so. Thus, we want to push the indirection down to the block RAMs. In other words, mesh connectivity information must not affect the DRAM access pattern and be entirely contained in the data layout and addressing of the

block RAMs. This is the argument for storing one mesh partitions in the block RAMs, using connectivity information to access it randomly without penalty and feeding the result into an arithmetic pipeline that will perform the floating point calculations. Under this arrangement, the edges are then represented as a vector of addresses into the node and cell RAMs.

3.2 Result accumulation and storage

Remember that res_calc performs an incrementing operation (also known as *reduction* with addition) on the *res* data set for the cells that each edge accesses. Therefore each edge computes only part of the final value of *res* of its cells. Therefore the arithmetic pipeline, upon processing each edge will produce increments that must be correctly summed up for each cell before that cell is output. Thus the result of the arithmetic pipeline must be added to the current value of res for the relevant cells. We use more block RAMs to store the intermediate *res* results, since the access pattern to *res* is the same as the access pattern for the cells.

The architecture diagram of that description is shown in Figure 3.1. The node and cell data for the current partition are streamed in from the DRAM and stored in the block RAMs. They are then read in a pattern described by the connectivity between edges, nodes and cells and fed into the pipeline that produces the *res* increments for two cells that must then be added to the currenty values of *res* for those cells. Thus we have an accumulator node between the block RAMs for *res* and the arithmetic pipeline.

The steps required to process a partition become:

1. Read in node and cell data from DRAM and store it locally.
2. Process the partition data and store the *res* data set locally.
3. Write out the resulting *res* set back to DRAM.

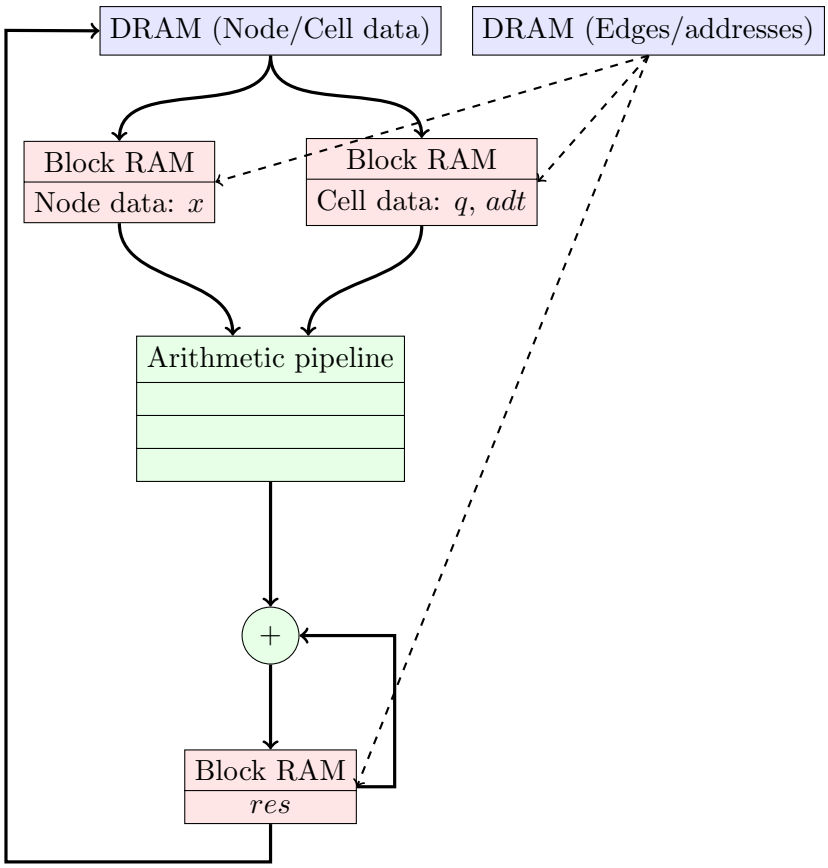


Figure 3.1: Simplified architecture diagram of the accelerator showing the block RAMs storing the node and cell data, the arithmetic pipeline, the result block RAMs and the accumulator. The connectivity information is used to address the block RAMs.

3.3 Halo exchange mechanism

In Airfoil every edge references two nodes and two cells. Those nodes and cells may be contained in the halo region of an adjacent partition. We have to devise a mechanism to acquire the required halo data. Since the mesh connectivity is constant, we can pre-compute the neighbours and the halo regions on the host. The difficulty arises from the reduction operation. A cell that can be accessed from two or more partitions needs to add up the contributions of all its edges, and the four edges that typically reference a

cell will not necessarily be in the same partition. We are faced with the problem of updating a cell from two or more partitons. Since we perform the initial partitioning, we can identify these halo cells and store them on the host, not on the DRAM of the card. When processing a partition, we will send these halo cells and nodes to the FPGA via PCIe. The accelerator will then have all the data it needs to process all its edges, however the *res* results that it computes for the halo cells will be only partial results that need to be combined with the results of the other partitions that access those cells. This addition of partial results will be performed on the host. We add some logic to choose the whether to read the cell and node data from the halo RAMs or the normal RAMs to obtain an architecture shown in Figure 3.2. This approach to halo exchange is similar to the ghost cell mechanism [8]. Thus the stages for processing a partition become:

1. Read in node and cell data from DRAM. Read in halo node and cell data from PCIe.

2. Process the partition data and store the *res* data set locally. The *res* data for the halo cells is a partial contribution of the final value.

3. Write the resulting *res* set back to DRAM. Send the partial results for halo cells back to the host through PCIe.

4. Once all the partitions are processed, the host adds up the contributions to the halo cells from all partitions.

Remember that the PCIe bandwidth is much lower than that of the DRAM (about 10 times lower), so if the halo region of a partition constitutes a large enough percentage of the total size of the partition, the PCIe transfer becomes dominant and the DRAM will end up being poorly utilised because the kernel will be waiting on the host transfer. This is a factor to keep in mind when choosing partition sizes and partitioning techniques.

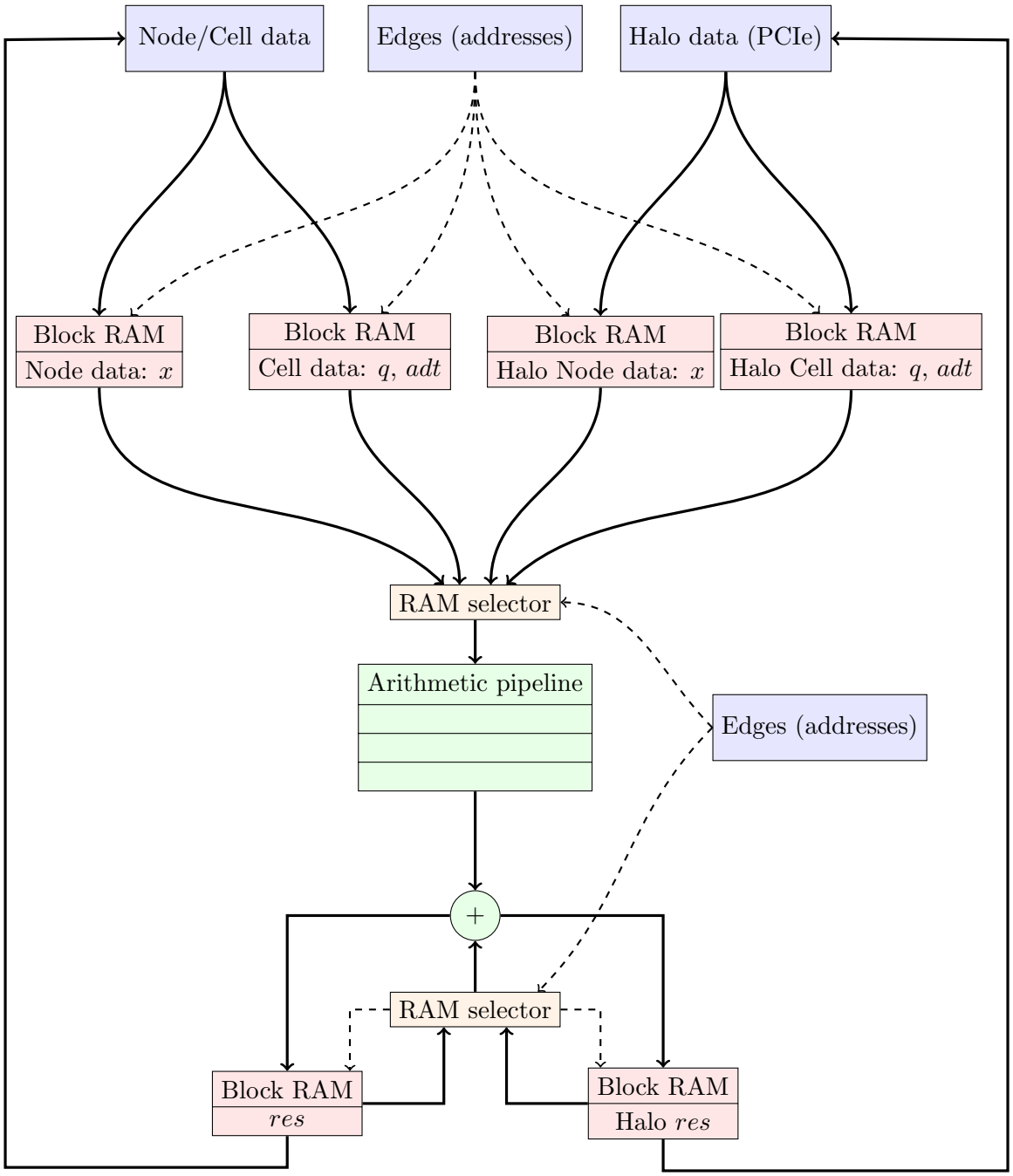


Figure 3.2: Architecture diagram of the accelerator with the PCIe halo exchange mechanism. The RAM selectors will select which RAM to read the cell and node data from based on the edge information. They are also used to pick the RAM to write the results back to. The dashed lines represent addresses that are used to access the RAMs and to determine which RAMs to access.

3.4 Two-level partitioning

The astute reader will notice that in order to process the partition, we first need to stream in the entire partition, process it fully and write it back. During the processing phase we are not streaming anything in or out of the kernel, leaving the DRAM unutilised, thus wasting bandwidth. To mitigate this, we introduce a second level of partitioning on the mesh. Each partition will be split into two *micro-partitions*(μpartitions). The idea is that as soon as we finish reading in the first mircro-partition, we can immediately start processing it while reading in the second micro-partition. Then, when the second mircro-partition has finished streaming in and the processing has finished for the first one, we can write out the results of the first one while processing the second. This allows us to achieve an overlap of data transfer and computation in a scheme reminiscent of a simple processor pipeline, where the fetch, computation and commit stages overlap to increase the utilisation of the relevant units. As shown in Figure 3.3, the two micro-partitions will invariably share some elements in a small region we call the *intra-partition halo*(IPH). Care must be taken to not write out the results of the intra-partition halo or overwrite the data in it before both mirco-partitions have finished processing. To avoid confusion we call the top-level partitions macro-partitions. This approach gives us the following phases in the accelerator:

1. Read in data for first micro-partition plus the intra-partition halo. If this is not the first macro-partition, write out the non-halo data for the second micro-partition and the intra-partition halo.

2. Process first micro-partition, read in the non-IPH data for second micro-partition.

3. Process second micro-partition, write out the non-IPH data for the first micro-partition.

Note that "read in" and "write out" include both the DRAM and PCIe halo transfers. This approach is expected to greatly improve DRAM utilisation at the expense of slightly more complex control logic. This imposes a constraint on the mesh layout in the DRAM and on the host machine. The first micro-partition should be stored before the intra-partition halo and the second micro-partition last.

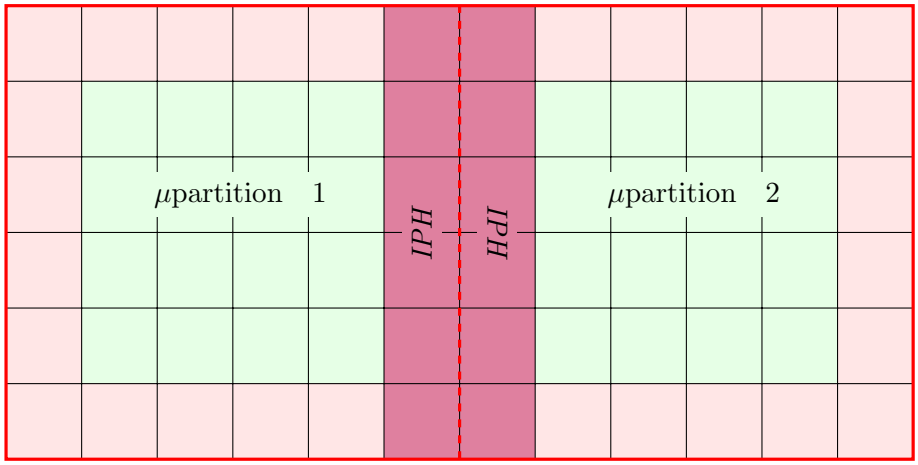


Figure 3.3: Partitioning of a macro-partition into two micro-partitions. This introduces a new intra-partition halo region, shown here in crimson.

The inputs, outputs and RAMs of the kernel can be controlled through enable signals that predicate their function on some boolean condition that we can define. This gives us a straightforward way to control when the kernel reads, processes or writes data. We can define a state machine with internal counters that can be used to keep track of the progress of each phase and signal the I/O units when data needs to be read in or written out. It can also be used to control the block RAMs, specifying when to commit the data found on their input ports. For this, the state machine will need to know the sizes of the micro-partitions, the intra-partition halo and the external halo. This can be added to the design as a separate stream that contains vectors of integers that represent the required sizes. Compared to the sizes of the partitions, the size of the size vector is negligible and therefore does not impact the performance of the memory system. Finally, we arrive at the architecture shown in Figure 3.4. The interleaving of I/O and processing gives rise to a pipeline-like execution pattern, shown in figure 3.5. Notice how at every stage there is I/O activity that keeps the DRAM and PCIe streams busy.

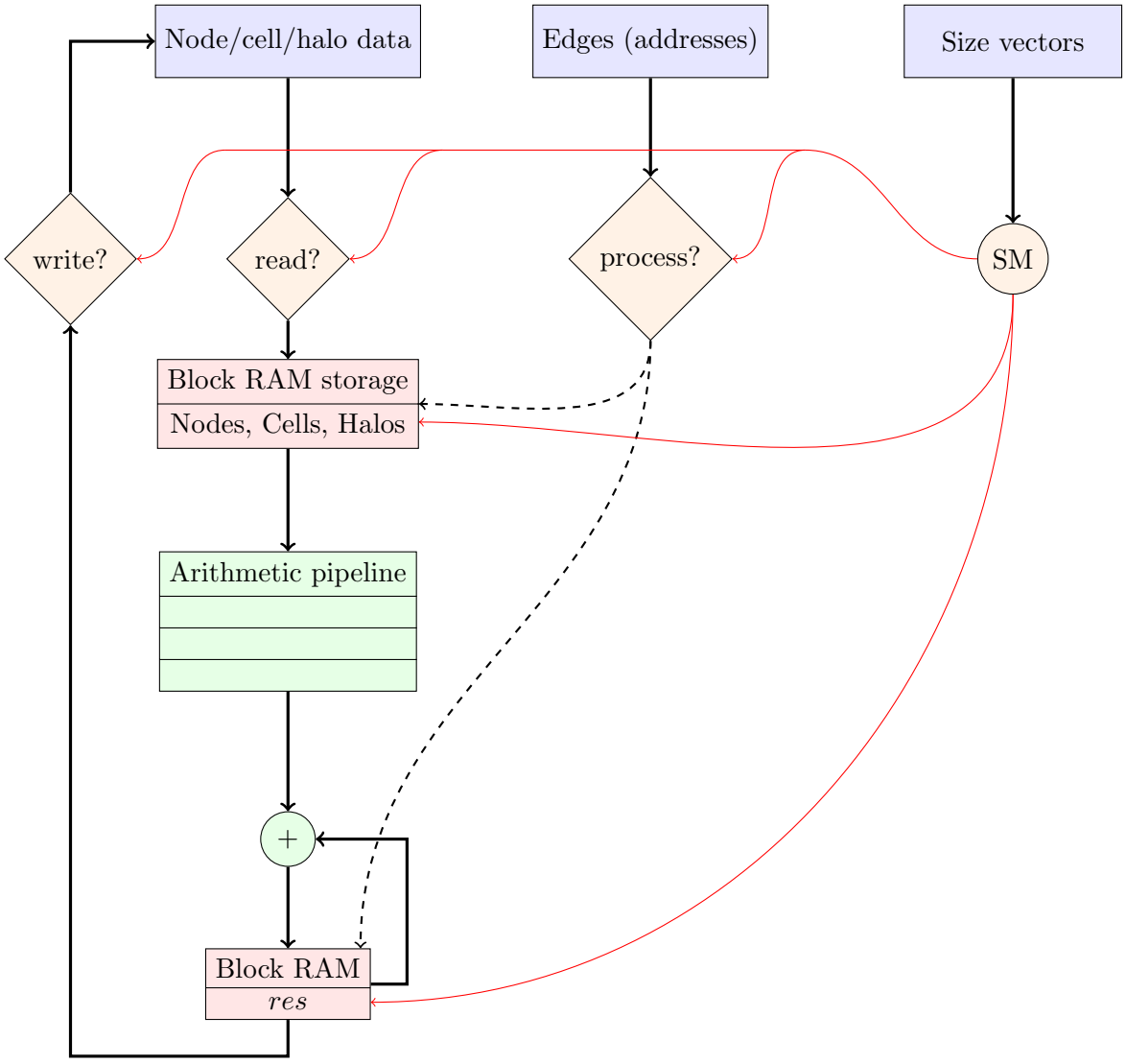


Figure 3.4: Architecture diagram showing the addition of a state machine (node SM) that controls the I/O and the processing. The red wires represent the boolean enable signals. The halo and normal RAMs as well as the RAM selectors are shown in merged blocks for brevity.

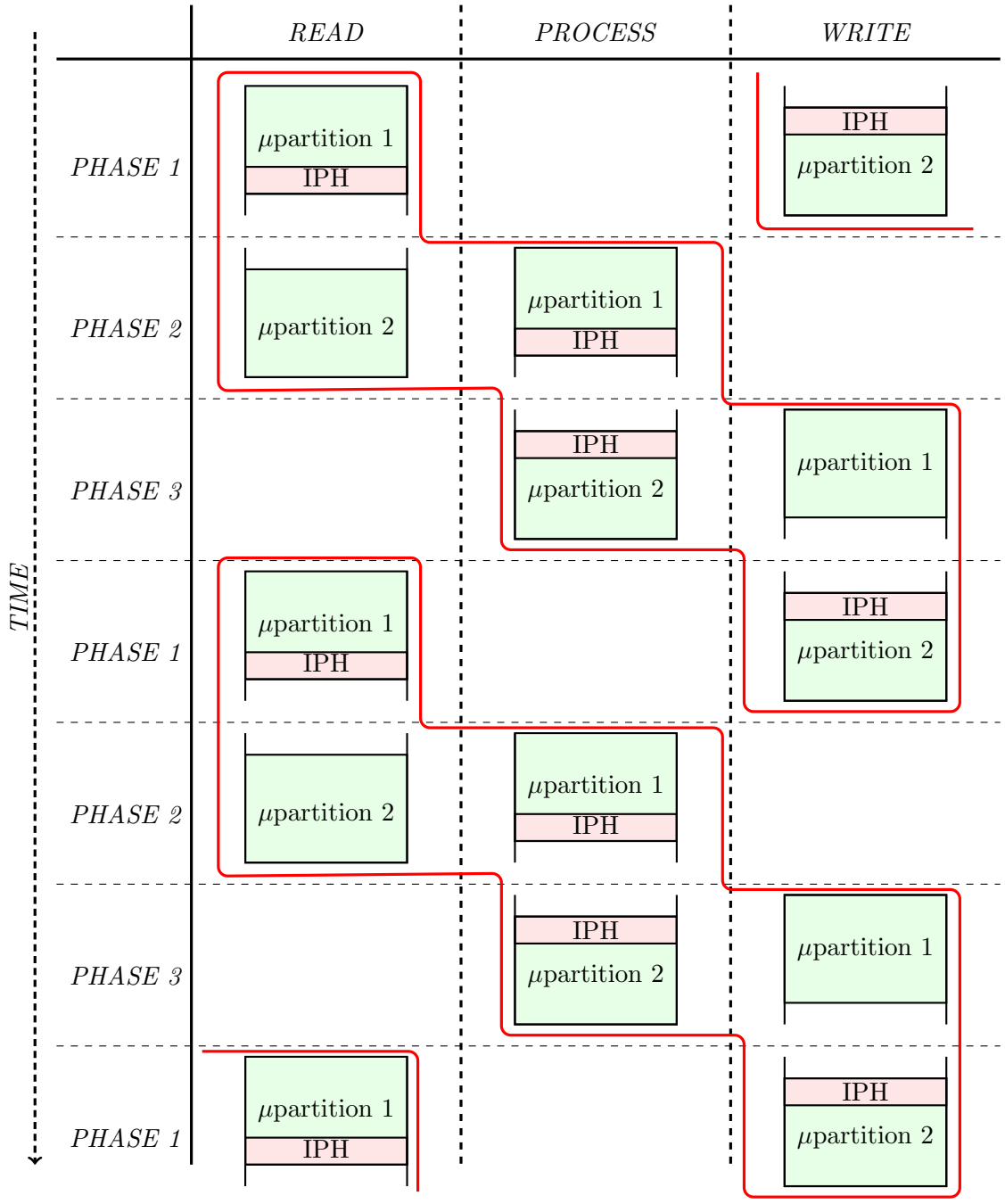


Figure 3.5: Diagram showing the overlapping of execution and I/O thanks to the two-level partitioning scheme. Note that both micro-partitions need the intra-partition halo (IPH) in order to be processed, so the IPH can only be written out together with the second micro-partition after both micro-partitions (μpartitions) have been processed. The red boxes represent the progress of a single (macro)partition through the accelerator phases.

3.5 The case for a custom streaming pipeline

The floating point calculations will be performed by the arithmetic pipeline, custom designed for the res_calc kernel. In a general purpose core, like on a CPU and to a somewhat lesser extent a GPU, the floating point calculations will be performed one after another, writing intermediate values to registers and/or cache. The calculations are expressed as a sequence of instructions. In a custom streaming datapath we specify a dataflow graph that the x, q and adt vectors are streamed through, and the res increments come out of the bottom. The advantage of this approach is that we can add registers at every stage of every calculation to create a deep pipeline with high throughput. Pipelining is a well-known processor design technique for increasing functional unit utilisation and throughput. On conventional processors it is used with some care, avoiding very deep pipelines, because the general purpose workload these CPUs are designed for may include arbitrary sequences of instructions that can potentially create various *pipeline hazards* (such as invalidation of an instruction already in a pipeline because a previous branch instruction was taken, a load memory instruction waiting on a write memory instruction etc.). Adding more pipeline stages may increases throughput (results per clock cycle) of instructions, but it also increases their latency (time for a particular instruction to complete) because of the extra registers that are added to store results between the pipeline stages. A pipeline hazard is usually dealt with by stalling the pipeline (waiting for an instruction to complete execution) or flushing it to remove invalid instructions. These measures introduce a performance penalty that is proportional to the depth of the pipeline [17].

These drawbacks are not applicable to our approach, because we are creating a custom datapath for a known custom workload that will perform specific floating point operations to data that is well-formed for this particular purpose. Since we know beforehand the calculations that will be performed and in what order we can safely pipeline the design as much as possible without worrying about data or control hazards. With that done, the only other concern becomes the task of keeping it occupied for as long as possible as discussed in the sections above.

An important and perhaps counter-intuitive prediction we can make is that because of the extreme pipelining the throughput of the architecture during the processing phase will remain at one result per cycle, regardless of the actual computational workload. As we increase the computational complexity of the kernel, the pipeline gets deeper and therefore takes more cycles to fill in the beginning and flush at the end, but during the time

when it's filled (which is most of the time if the architecture works properly and supplies inputs continuously) it produces one result per clock cycle. This gives us an intuition of why this approach to acceleration is a good idea. Assuming large enough data sets, as in Airfoil, the time to fill and flush the pipeline will be amortised by the time spent executing. Compare this observation with the execution of Airfoil on a CPU or a GPU. On those architectures the arithmetic execution time is expected to increase proportionally to the number of floating point calculations performed. Using a custom streaming datapath, the increase in floating point calculations translates into more resources being used (LUTs, Flip Flops and DSPs), but does not imply a corresponding increase in execution time. This means that a custom, deeply pipelined architecture must win in the asymptotic case against any general purpose architecture as the number of arithmetic operations increases. Of course, in practice, the complexity of the arithmetic pipeline will be limited by the amount of resources available on the chip.

Figure 3.6 shows the difference in the custom streaming approach as opposed to a conventional CPU. Note the absence of a fetch/decode unit in the custom approach, since we are not dealing with instructions. The arithmetic pipeline is designed to implement a particular mathematical function, while on a generic CPU, the Arithmetic and Logic Unit (ALU) can handle arbitrary sequences of arithmetic operations, at the expense of requiring a separate fetch/decode unit as well as a register file to store intermediate results. Notice how the `add r1, r2` instruction cannot execute until the previous instruction `mul r1, r1` has commited its result to register `r1`, creating a potential data hazard that may stall any pipelines present in the functional units of the ALU. In the streaming approach on the left, each of the functional units is pipelined to achieve high throughput. The y stream must be delayed by a FIFO queue before being sent to the adder to compensate for the cycles taken to produce the result from the multiplication unit. Similarly, the output of the square root unit must be delayed/buffered before entering the subtraction node. This way, we can push new values for x, y and z into the pipeline every cycle and after it has been filled, we start receiving one value for r every cycle. Notice, also, that the x^2 and $\sqrt{z}$ functions are computed in parallel, since they operate on different data items. Compare this to the CPU approach on the right, where the computation of a single value for r takes 8 instructions. It is evident that even with sophisticated processor design techniques like out-of-order execution, value forwarding and instruction reordering by the compiler, it is unlikely that the CPU will be able to produce and sustain a throughput of one value per clock cycle since the `load` instructions alone will probably take at least

one cycle to fetch the data from the cache or, even worse, the main memory in the event of a cache miss. This should be especially evident on large homogeneous workloads where the time to fill and flush the custom pipeline on the left becomes negligible compared to the time it remains filled, providing maximal throughput, while the CPU case will have to deal with more cache misses that introduce huge performance penalties (in the 1000s of clock cycles) due to the fact that the large data sets will simply not fit into the cache. Of course, the custom streaming approach also demands that data be fed into the pipeline at every cycle, thus requiring additional effort by the developer to format the data accordingly as discussed in previous sections.

A potential GPU implementation will also suffer from these problems, albeit to a lesser degree. While a GPU has hundreds of cores and arithmetic units available to run hundreds or thousands of threads, they are still general purpose and still suffer from the need to decode instructions and access a register file and a cache/memory hierarchy. Adding more complex arithmetic operations will still increase the time to produce a result in a linear fashion, while in a custom streaming datapath the extra work can either be done in parallel by adding extra function nodes and streams or by adding more nodes to the pipeline, increasing the time to fill/flush it, but maintaining the throughput.

The arithmetic pipeline we design for the res_calc kernel will produce increments that must be added to the previous values of *res* by the accumulator, as discussed earlier. We choose to use single precision IEEE-754 floating point numbers for the arithmetic because previous work by Sanchez-Roman et al. [11] hints at the error accumulation that arises from repeated iterations of fixed-point calculations as well as the implementation difficulties in encoding the data sets to and from fixed-point representation due to the fact that most host CPU architectures do not have native fixed-point data types. Trial runs of the serial version of Airfoil on a CPU have shown the iteration structure to converge when using single precision floating point numbers so we do not use double precision arithmetic, since this is a proof of concept project.

Note that the res_calc arithmetic pipeline has too many nodes to be meaningfully reproduced in a diagram here, but it is constructed using the principles discussed in this section.

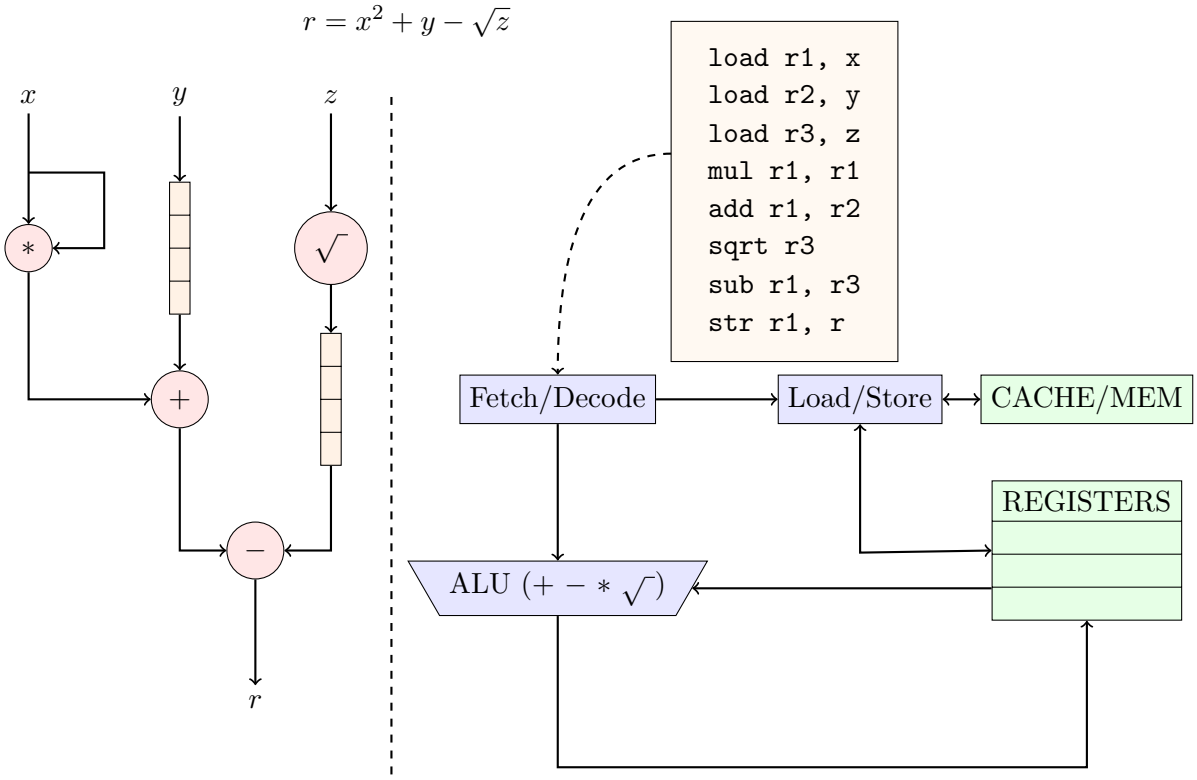


Figure 3.6: Diagram showing the computation of the function $r = x^2+y-\sqrt{z}$ by using a custom streaming datapath (left) and using a sequence of instructions in a conventional CPU (right).

3.6 Performance Model

In order to justify the design decisions made, we present a performance model that aims to predict the maximum achievable performance increase. For this we need to take into account the hardware characteristics, such as DRAM and PCIe bandwidth and the chip clock frequency as well as the format of the data, in particular the widths of the data sets. We start by considering the structure and partitioning of the mesh. In the sample meshes that we have the number of nodes is roughly the same as the number of cells. The number of edges is about twice the number of cells/nodes.

Let C_{tot} = total number of cells, N_{tot} = total number of nodes, E_{tot} = total number of edges. C_{pp} is the number of cells per partition and depends solely on the amount of memory available on the FPGA. Same

goes for the number of nodes per partition N_{pp}. N_{pp} is typically equal to C_{pp}. We use this number to specify the desired partition size to the METIS partitioning tool to get n partitions. For the purposes of this analysis, we assume that the partitions are roughly square in shape. By square, we mean that they have an equal number of cells/nodes on their sides/borders. In practice the tools may deliver arbitrary shaped partitions. Assuming square partitions of dimension $l \times l = C_{pp}$, the number of halo cells will be $C^h = l + l + 2 \times (l - 1) \approx 4l = 4 \times \sqrt{C_{pp}}$ as l becomes large. The same for the number of halo nodes $N^h = 4l = 4 \times \sqrt{C_{pp}}$. The number of edges per partition will be $E_{pp} = E_{tot}/n$.

We can reasonably assume that partitioning the macro-partitions into two micro-partitions will give two micro-partitions of equal size. In that case the number of cells per micro-partition is $C_{\mu p} = C_{pp}/2$ and the number of nodes is $N_{\mu p} = N_{pp}/2$ and similarly for the edges $E_{\mu p} = E_{pp}/2$. In a similar fashion we calculate the numbers for the halo regions of each micro-partition: $C^h_{\mu p} = C^h/2$, $N^h_{\mu p} = N^h/2$. We assume the intra-partition halo to be negligible in size, which is confirmed in practice. We represent the width of the data sets involved in res_calc in bits: $\overline{q} = 4 \times 32 = 128$, $\overline{x} = 2 \times 32 = 64$, $\overline{adt} = 32$, $\overline{res} = 4 \times 32 = 128$.

Next, we take into account the hardware characteristics. The important factors are: the DRAM bandwidth (B_D), the PCIe bandwidth (B_P) and the clock frequency f. Note that the DRAM bandwidth B_D is shared between streams to and from DRAM, while the PCIe streams get B_P of bandwidth to the FPGA and B_P from the FPGA.

Recall the phases of computation in the accelerator:

1. Read in data for first micro-partition plus the intra-partition halo. If this is not the first macro-partition, write out the non-halo data for the second micro-partition and the intra-partition halo.

2. Process first micro-partition, read in the non-IPH data for second micro-partition.

3. Process second micro-partition, write out the non-IPH data for the first micro-partition.

The total execution time can be estimated by adding the execution times for each of the three phases. So we consider each phase in turn.

3.6.1 Phase 1

The size of the input data for one cell is $\overline{C_{in}} = \overline{q} + \overline{adt} = 128 + 32 = 160$ bits. The size of the input data for one node is just the size of the x dataset: $\overline{x} = 64$. Therefore the amount of data that needs to be transferred from DRAM is $D^{in}_{DRAM} = C_{\mu p} \times \overline{C_{in}} + N_{\mu p} \times \overline{x}$. We are also writing back the res vectors back to DRAM and PCIe from the previous micro-partition. The width of the result is $\overline{C_{out}} = \overline{res} = 128$ and thus the amount of data to be written out to DRAM is $D^{out}_{DRAM} = C_{\mu p} \times \overline{C_{out}}$. Similarly the amount of halo data to be transferred from PCIe is $D^{in}_{PCIe} = C^{h}_{\mu p} \times \overline{C_{in}} + N^{h}_{\mu p} \times \overline{x}$ and the data written out to PCIe is: $D^{out}_{PCIe} = C^{h}_{\mu p} \times \overline{C_{out}}$. The DRAM bandwidth is shared among both input and output streams, therefore the bandwidth allocated to the input stream from DRAM to the kernel will be $B^{in}_{D} = B_D \times \frac{D^{in}_{DRAM}}{D^{in}_{DRAM} + D^{out}_{DRAM}}$ and the bandwidth to DRAM will be $B^{out}_{D} = B_D - B^{in}_{D}$. As mentioned earlier, both PCIe streams get the same bandwidth B_P. Knowing the bandwidths we can calculate the times needed for the data transfers. $t_{DRAM} = \frac{D^{in}_{DRAM}}{B^{in}_{D}} = \frac{D^{out}_{DRAM}}{B^{out}_{D}}$ for the DRAM transfer and $t_{PCIe} = \frac{D^{in}_{PCIe}}{B_P}$ for the PCIe transfer since the width of the input data is greater than the width of the output data and the bandwidths for input and output are the same. The FPGA itself is a synchronous circuit operating at f cycles per second. This means that if it wants to read a data item from but the stream does not have anything available the kernel will stall until the memory system supplies a data item. Since the PCIe channel is slower than the DRAM channel, we need to make sure that the PCIe accesses happen at regular intervals and not one after the other so as to avoid forcing the DRAM to deliver data in lockstep with the PCIe bus. Similarly for output streams. We assume such a configuration here. We assume that we read/write one node/cell pair from DRAM every cycle and a halo node/cell pair from PCIe every few cycles in such a way as to not throttle the DRAM. Assuming a clock frequency f, the minimum time needed to consume the data for a micro-partition will be $t_{FPGA} = \frac{C_{\mu p}}{f}$. Thus, the time taken to complete phase 1 will be the maximum of the three times calculated: $t_1 = max(t_{DRAM}, t_{PCIe}, t_{FPGA})$. Ideally we want the three times to be in then order: $t_{PCIe} < t_{DRAM} < t_{FPGA}$, i.e. we don't want PCIe transfer to be a bottleneck.

3.6.2 Phase 2

In phase 2, the first micro-partition gets processed, i.e. its edges are streamed in, used to address the block RAMs and results are written to the *res* block RAMs. Also, the second micro-partition is streamed in and stored in local storage. Again, we stream in data of width $\overline{C_{in}}$ from both DRAM and PCIe, but this time we also stream in edge data in the for of four addresses: one for each of the two cells and nodes. The width of the edge data depends on the number of elements that are in a partition. A block RAM of depth/size d has addresses of width $\lceil log_2(d) \rceil$ bits. The depth of the block RAMs in our case is the number of cells/nodes in a partition C_{pp} (typically, the number of nodes and cells is roughly equal), therefore the width of each address is $\lceil log_2(C_{pp}) \rceil$ and the width of the data for each edge is $\overline{E_{in}} = 4 \times \lceil log_2(C_{pp}) \rceil$. Therefore the amount of data we are transferring from DRAM will be $D^{in}_{DRAM} = E_{\mu p} \times \overline{E_{in}} + C_{\mu p} \times \overline{C_{in}} + N_{\mu p} \times \overline{x}$, while the amount of halo data transfered from PCIe will be $D^{in}_{PCIe} = C^{h}_{\mu p} \times \overline{C_{in}} + N^{h}_{\mu p} \times \overline{x}$.

As before, we calculate the times to transfer the data from DRAM $t_{DRAM} = \dfrac{D^{in}_{DRAM}}{B_D}$ and PCIe $t_{PCIe} = \dfrac{D^{in}_{PCIe}}{B_P}$.

This time, the number of cycles taken to consume the data by the kernel will be dominated by the number of edges, since there are about twice the number of edges on average than cells or nodes. The number of edges we process per cycle is the number of arithmetic pipelines we have n_p. The minimum time taken by the FPGA to consume all edges in a micro-partition in phase 2 will then be $t_{FPGA} = \dfrac{E_{\mu p}}{f \times n_p}$. Again, the total runtime for this phase will be dominated by the maximum of the three times calculated above, that is $t_2 = max(t_{DRAM}, t_{PCIe}, t_{FPGA})$.

3.6.3 Phase 3

The third phase is similar to the second one, except that we are now writing back the first micro-partition and processing the second one. The numbers remain the same for t_{DRAM}, t_{PCIe} and t_{FPGA}. $t_3 = max(t_{DRAM}, t_{PCIe}, t_{FPGA})$. The total time to process one iteration of the whole mesh is therefore: $t_{tot} = n \times (t_1 + t_2 + t_3)$. The time to execute i iterations then becomes $t^i_{tot} = i \times n \times (t_1 + t_2 + t_3)$.

3.6.4 Design space exploration

Having the above model, we can plug in values for existing test meshes and architecture options to explore the effects that each one will have on the performance. This way we can identify bottlenecks in performance and optimise the architecture accordingly. Substituting values for our test mesh of 721801 nodes, 720000 cells and 1438600 edges and $2^13 = 8192$ cells/nodes per partition for an architecture with one arithmetic pipeline running at $f = 240$MHz we get values for $t_1 = 1.71 \times 10^{-5}s, t_2 = 3.44 \times 10^{-5}s, t_3 = 3.44 \times 10^{-5}s$ for a total runtime $t_{tot} = 7.48 \times 10^{-3}s$. Extrapolating to 2000 iterations, we get $t_{tot}^{2000} = 14.96s$. This estimated runtime beats the time of 53.99 seconds for res_calc shown in Table 3.1, achieving a speedup of $\times 3.6$. However, it is more interesting to compare the runtime to an accelerated version. One such implementation, developed at the Software Performance Optimisation group at Imperial College uses the NVIDIA Tesla M2050 accelerator [19] programmed using the OpenCL framework [18]. It uses the large number of cores present on the card to launch thousands of threads in parallel, thus greatly accelerating computation. The times for the various Airfoil kernels on the same test mesh are shown in Table 3.1:

Kernel Name	Time spent (seconds)	Percentage of total time (%)
save_soln	0.4593	4.75
adt_calc	1.1310	11.69
res_calc	6.5122	67.34
bres_calc	0.0640	0.66
update	1.5042	15.55

Table 3.1: Table showing the time spent in each kernel during a run of a hardware accelerated version of Airfoil on a Tesla M2050 GPU. The total run time is 9.67 seconds.

Notice how the execution time has improved in contrast to the single-threaded CPU variant and how res_calc now dominates the execution time. The time we should be planning to beat is 6.51 seconds. In the graphs following, the times are reported in seconds.

Number of arithmetic pipelines n_p

If we look at the numbers we computed for the above mesh we notice that for phases 2 and 3 of the accelerator, the time spent computing the edges (t_{FPGA}) is $3.44 \times 10^{-5}s$, dominating the execution of that phase, while

the time for DRAM transfer is $t_{DRAM} = 4.43 \times 10^{-6} s$. This is a strong indication that the edge processing is the bottleneck in performance. Recall that $t_{FPGA} = \frac{E_{\mu p}}{f \times n_p}$. $E_{\mu p}$ is a characteristic of the mesh and not subject to variation. The clock frequency f and the number of arithmetic pipelines n_p we can vary. Lets start with n_p. Increasing the number of pipelines we get a considerable boost in performance, as shown in Figure 3.7.

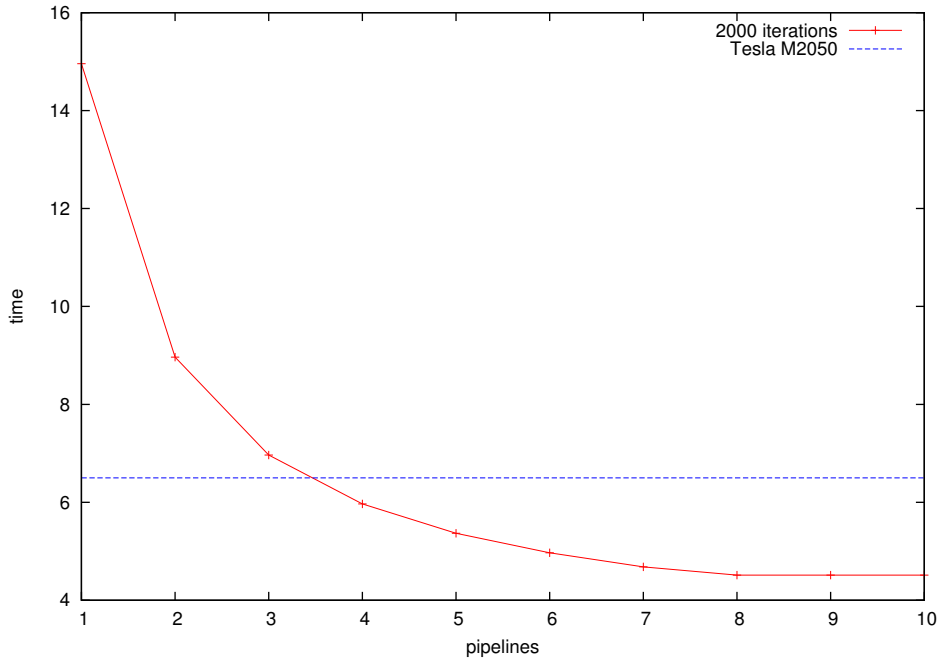


Figure 3.7: Plot of estimated execution time for 2000 iterations of res_calc against the number of arithmetic pipelines in the architecture. The clock frequency is set to 240MHz.

Notice how after 8 pipelines we do not get a performance boost anymore because the bottleneck now shifts to the memory transfers, DRAM in particular. Using just 4 pipelines our architecture is estimated to beat the Tesla performance, which is an impressive result, considering the hundreds of concurrent cores running on the Tesla.

Clock frequency f

Next we explore the effects of varying the clock frequency of the chip. Realistically, the frequencies achievable on the hardware range from 120MHz to 300MHz. The results of increasing the frequency are presented in Figure 3.8. While frequencies above 300MHz are not in general achievable, we present the theoretical increase it would take to catch up to the Tesla implementation. As we can see, raising the clock frequency is clearly a worse choice than increasing the number of pipelines. Note also that raising the clock frequency implies more energy consumption leading to more heat dissipation, further reducing the temptation to raise it.

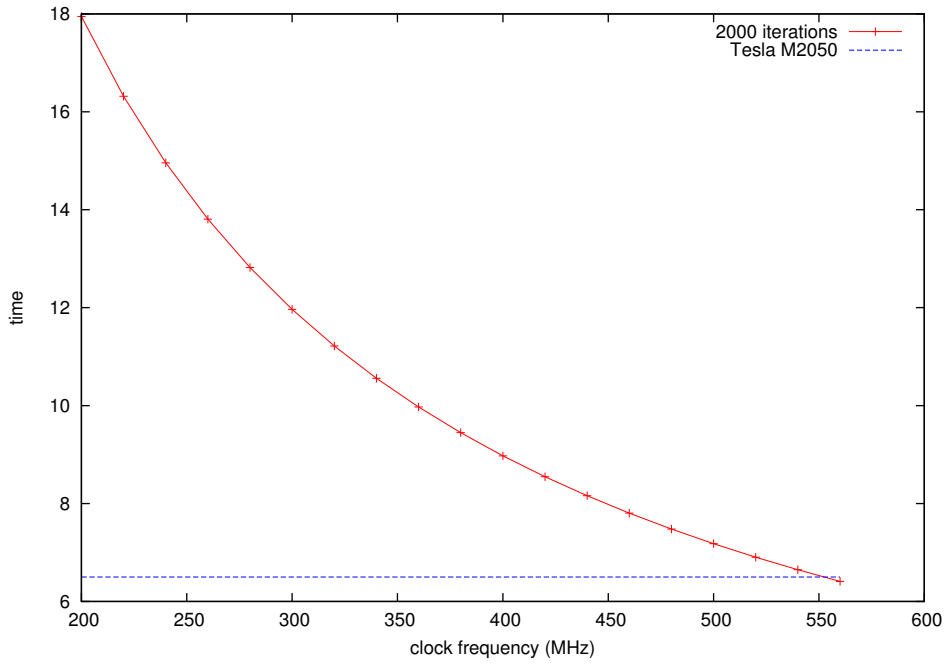


Figure 3.8: Plot of estimated execution time for 2000 iterations of res_calc against the clock frequency of the FPGA using one arithmetic pipeline.

Clock frequency f and arithmetic pipelines n_p

It is interesting to see the effects of varying the frequency f as well as the number of arithmetic pipelines n_p. The results of such an experimentation are shown in Figures 3.9 and 3.10.

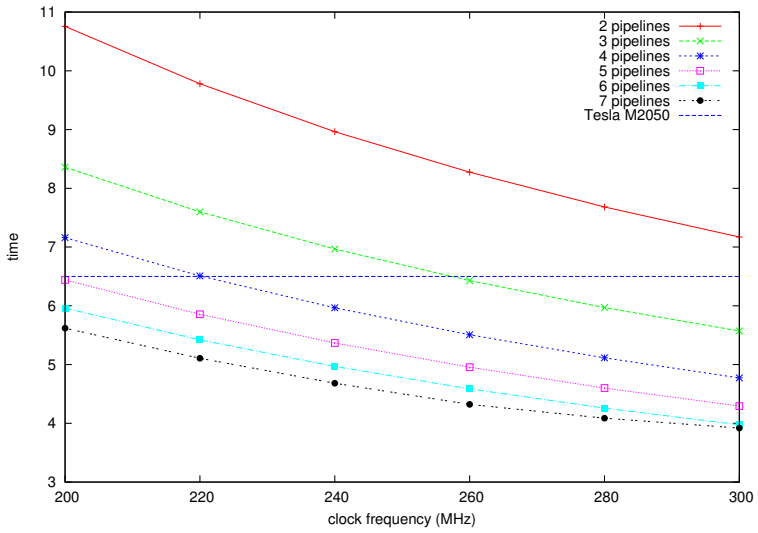


Figure 3.9: Plot of estimated execution time for 2000 iterations of res_calc against the clock frequency of the FPGA for various numbers of arithmetic pipelines.

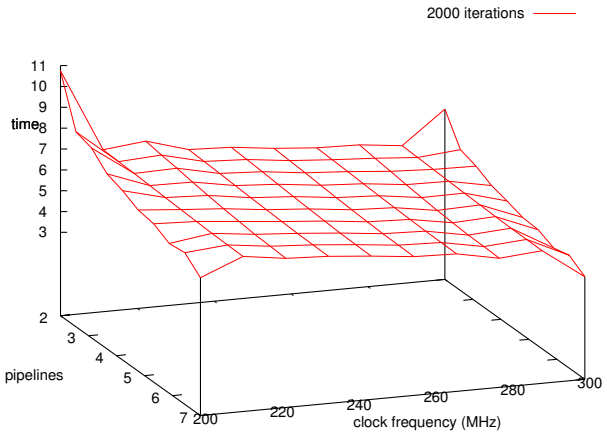


Figure 3.10: Plot of estimated execution time for 2000 iterations of res_calc against the clock frequency of the FPGA for various numbers of arithmetic pipelines in a 3-dimensional plot.

We see that by using 6 or 7 pipelines and clocking the design at 300MHz, we can beat the Tesla implementation by about 66%, offering a total of $\times$13.8 speedup over the serial CPU version. These numbers validate the architecture we designed by providing some estimates about the potential speedup. Having performed this performance analysis, we can now move on to implementing the architecture in order to discover potential issues that might prevent us from achieving the maximum performance that we predict with this model.

Partition size C_{pp}

Once we remove the arithmetic pipeline bottleneck by adding more pipelines we can experiment with partition sizes to see the effects. varying the number of cells per partition C_{pp} gives us the data shown in Figure 3.11. Note, also the ratio of non-halo to halo transfers in Figure 3.12.

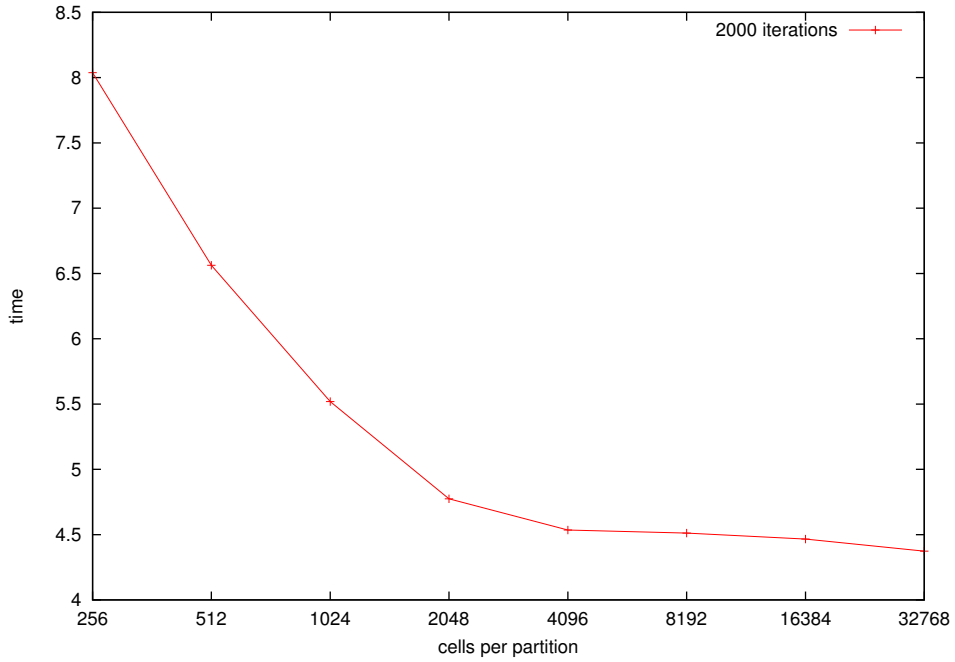


Figure 3.11: Plot of estimated execution time for 2000 iterations of res_calc against the number of cells per partition. Note that the horizontal axis is on a log-2 scale. The design has 8 pipelines and runs at 240MHz.

We notice that for values of C_{pp} less than 2048 the PCIe transfer time

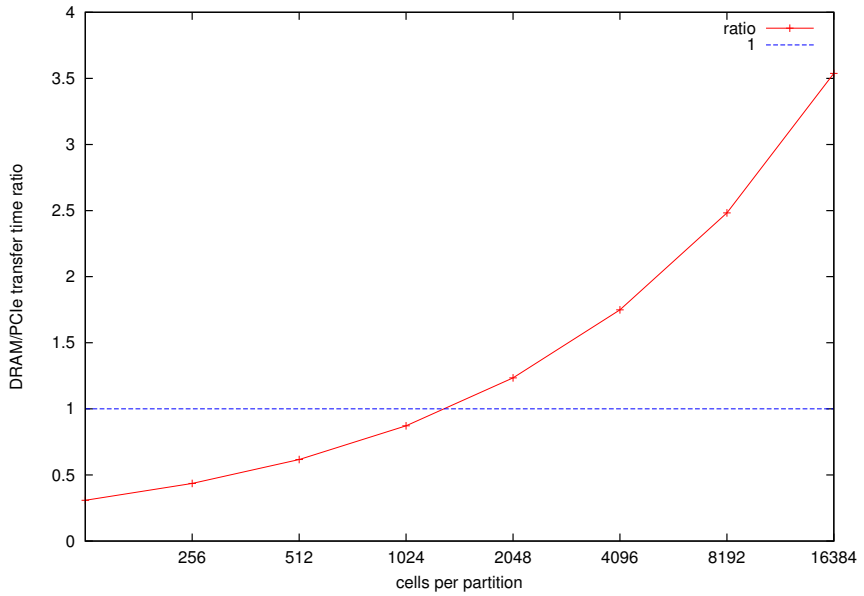


Figure 3.12: Plot of estimated DRAM to PCIe transfer time ratio ($\frac{t_{DRAM}}{t_{PCIe}}$) against the number of cells per partition. Note that the horizontal axis is on a log-2 scale.The design has 8 pipelines and runs at 240MHz.

dominates execution time. Notice how steeply the performance degrades when PCIe transfers dominate. This result is to be expected since the size of the halo region is proportional to the square root of the partition size, in other words $C^h_{pp} \propto \sqrt{C_{pp}}$.

Chapter 4

Implementation

In this chapter we present the issues that arise when trying to implement the FPGA-based accelerator and the host-side mesh preprocessing code. Due to time limitations we attempt to build a design using one arithmetic pipeline with 8192 (2^13) cells per partition.

4.1 Mesh partitioning

We use the METIS partitioning tool through its C API to assign a partition number to each cell and node. This is where we use the indirection maps that define the connectivity. When assigning edges to partitions we must take care to also add the nodes and edges that they reference from neighbouring partitions. These nodes and cells will be part of the halo region.

Next, we need to perform the second level of partitioning. For this we need to translate the global indirection maps (*cell, edge* etc) into local maps for each partition. For this we need to remap the nodes and cells to a local numbering. We use hash maps to store the associations between global and local numbering and use them to translate the *cell* indirection map into local maps for each partition.

Now we can perform the second level of partitioning by breaking up each partition into 2 micro-partitions. We actually want three regions from this process: the two micro-partitions and the intra-partition halo (*IPH*). We assign edges to each partition again taking care to add the nodes and cells from the neighbouring micro-partition. Since METIS provides a partition number for each cell and node, we need to Figure out the IPH region ourselves. We store the nodes for each micro-partition into a set (we used a hash set but any implementation could be used) called $Nodes_{\mu p1}$ and $Nodes_{\mu p2}$

and calculate the IPH region by $Nodes_{IPH} = Nodes_{\mu p1} \cap Nodes_{\mu p2}$. Now we need to remove the common nodes from $Nodes_{\mu p1}$ and $Nodes_{\mu p2}$, that is $Nodes_{\mu p1} := Nodes_{\mu p1} \backslash Nodes_{IPH}$ and $Nodes_{\mu p2} := Nodes_{\mu p2} \backslash Nodes_{IPH}$, where $A \backslash B$ is the set A with the elements of set B removed ($\backslash$ is the set difference operator). We follow the same procedure for cells and we arrive at the partitions that we need.

4.2 Edge scheduling

Recall the use of the accumulator that adds the increments computed by the arithmetic pipeline to the current values of res (figure 3.1), creating a loop in the architecture. This has implications on the order with which we process the edges. Suppose the accumulator has latency l, that is if the operands for addition arrive to it at cycle c the corresponding result will appear on its output at cycle $c + l$. That latency depends on the number of pipeline stages in the adder. One of the operands is the increment computed by the arithmetic pipeline and the other is the previous value of res that was read from the corresponding BRAM. The new res result that must be written back to the BRAMs will be produced in l cycles. This means that within that window we must not access the same BRAM address (i.e. the same cell) because we will have not commited the previous result of res that is still in the accumulator pipeline, thus getting two values for res that will be commited to the BRAM with the earliest one being overwritten instead of being added. This outcome is shown in Figure 4.1 where two values v_1 and v_4 for the cell at α_0 are being computed. When v_4 exits the pipeline it will be written at address α_0 in the BRAM, only to be overriden by v_1 3 cycles later. To work around this issue we constrain the iteration order of the edges in each micro-partition. The constraint is that for every edge no edge in a window of width l must access the any of the two cells. Thus, given an ordering of n edges sch the validity of the schedule sch for a window width l can be determined by algorithm 1.

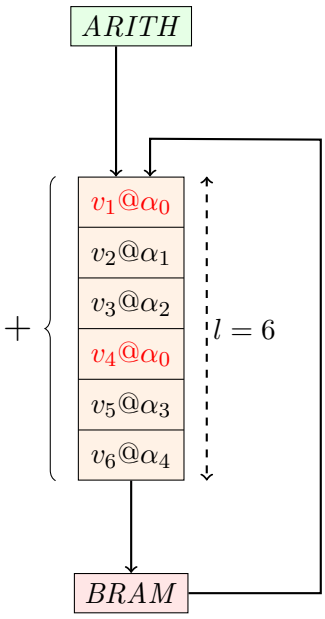


Figure 4.1: Architecture diagram of the accumulation part of the accelerator design showing the conflicting values of *res* being computed in the accumulator pipeline. The conflicting values for address α_0 are shown in red.

```
function boolean validSchedule(node[ ]sch, int n, int l)
    for i in [0..n − 1] do
        C_i ← set of cells referenced by edge sch[i]
        for j ← 1 ; j < l ; j ← j + 1 do
            C_(i+j)%n ← set of cells referenced by edge sch[(i + j)%n]
            if C_i ∩ C_(i+j)%n ≠ ∅ then
                return FALSE
            end if
        end for
    end for
    return TRUE
end function
```

Algorithm 1: Pseudocode that validates an edge schedule *sch* of n edges with window width l.

In order to compute that schedule we partition each micro-partition

again into multiple partitions, calling them *edge partitions*. We define two edge partitions being *adjacent* if an edge in one references a cell in the other. Using this definition we construct an *adjacency graph* where the nodes are the edge partitions and we connect two nodes with an edge if the two edge partitions are adjacent. That way we know that edges belonging to two non-adjacent partitions will never conflict and can be scheduled one after the other. That way, the problem of finding a schedule for the edges can be reduced to finding a schedule for the nodes in the adjacency graph such that no two nodes within a window of width of width l will be adjacent. An example is shown in Figure 4.2. Using the adjacency graph we can transform Algorithm 1 for checking the validity of a schedule into Algorithm 2.

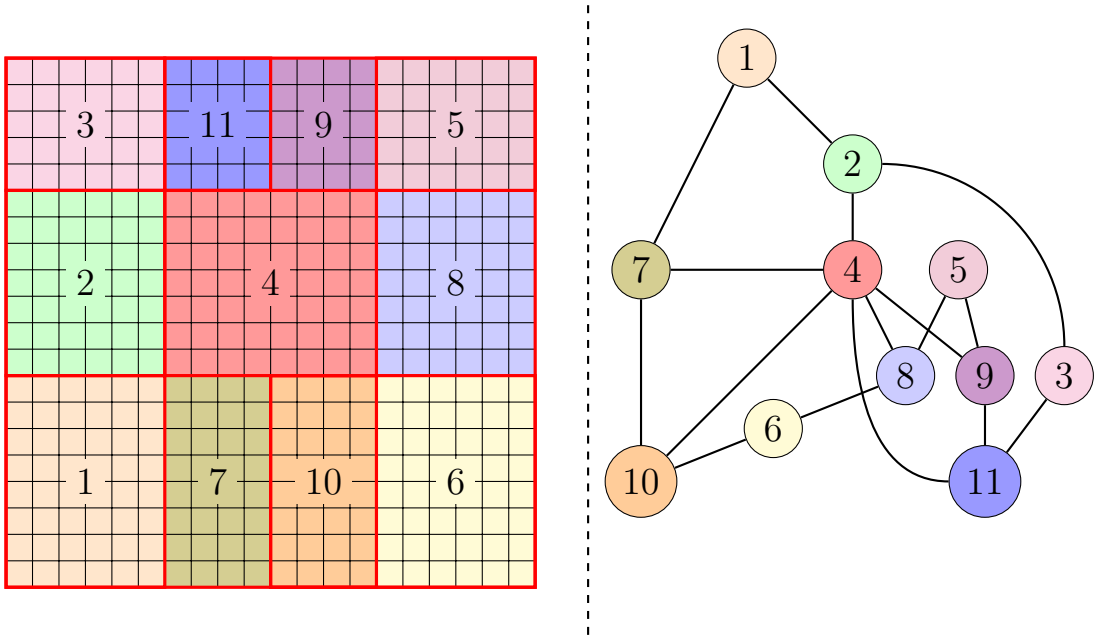


Figure 4.2: An example partitioning of a micro-partition into 11 edge partitions (left) and the adjacency graph generated from that partitioning (right).

```
function boolean validSchedule(node[ ] sch, int n, int l, Graph g)
    for i in [0..n − 1] do
        for j := 1 ; j < l ; j := j + 1 do
            if sch[i] adjacent to sch[(i + j)%n] in g then
                return FALSE
            end if
        end for
    end for
    return TRUE
end function
```

Algorithm 2: Pseudocode that validates a node schedule *sch* of an adjacency graph g with n elements for a window width of l.

To actually compute a schedule given an adjacency graph and a window width, we can use algorithm 3. Note that the *sch* function is recursive and uses the *validPos* function defined in algorithm 4 to pick which nodes to attempt to schedule. Due to the high number of adjacency checks, we choose to represent the graphs as two-dimensional adjacency matrices *mat* where $mat[i][j] = 1$ if nodes i and j are adjacent and $max[i][j] = 0$ otherwise. Algorithm 3 performs a depth-first search of the state space, backtracking to try different paths if a particular path fails thanks to the use of queues to store nodes that should be tried next. The recursion handles the backtracking. At every call of *sched* the algorithm will try to place a node into the schedule and determine which of its non-adjacent nodes are eligible for consideration, calling itself recursively to place the next node.

```
function node[ ] scheduleGraph(Graph g, int l)
    nnodes ← number of nodes in g
    res ← node[nnodes]                          ▷ array of size nnodes
    count ← 0
    q ← empty Queue
    for all node in g do
        enqueue node in q
    end for
    while q not empty do
        n ← q.dequeue()
        if sch(n, &count, &res, g,) then
            return res
        end if
    end while
    print("Could not schedule graph")
    return NULL
end function
function boolean sched(node n, node[ ] res, int count, Graph g, int l)
    res[count] ← n
    count ← count + 1
    q ← empty Queue
    if count = number of nodes in g then
        return TRUE
    end if
    for all nn in g and not adjacent to n do
        if nn ∉ res ∧ validPos(res, nn, count, l, g) then
            enqueue nn in q
        end if
    end for
    while q not empty do
        child ← q.dequeue()
        if sched(child, &res, &count, g, l) then
            return TRUE
        end if
    end while
    count ← count − 1          ▷ Failed to find schedule down this path
    return FALSE
end function
```

Algorithm 3: Pseudocode that computes a schedule for the nodes in a graph g with window width l.

```
function boolean validPos(node[ ] arr, node n, int c, int l, Graph g)
    if c ≤ l then
        for p ← 0 ; p < count ; ++p do
            if arr[p] adjacent to n in g then
                return FALSE
            end if
        end for
    end if
    nnodes ← number of nodes in g
    if c > nnodes − l then
        for p ← 0 ; p < l − (nnodes − c) ; p ← p + 1 do
            if arr[p] adjacent to n in g then
                return FALSE
            end if
        end for
    end if
    for p ← c − l − 1 ; p < c ; p ← p + 1 do
        if arr[p] adjacent to n in g then
            return FALSE
        end if
    end for
    return TRUE
end function
```

Algorithm 4: Pseudocode that checks whether inserting node n at position c into the schedule arr will produce a valid partial schedule for window width l of the nodes in graph g.

4.2.1 No-op edges

Once the edge-partition schedule has been computed, the edge order can be computed by taking an edge from each partition in the scheduled order in turn. However, in practice each edge-partition will not contain exactly the same amount of edges, leaving "gaps" in the pipeline. To work around this, we add *no-op edges*(no operation edges), dummy edges that will be used to fill the gaps in the pipeline and whose resulting res increments will be ignored by the accelerator commit phase. The conversion from an edge-partition schedule to a schedule for the edges is shown in Figure 4.3. No-op edges clearly are a necessary but undesirable addition to the edge schedule

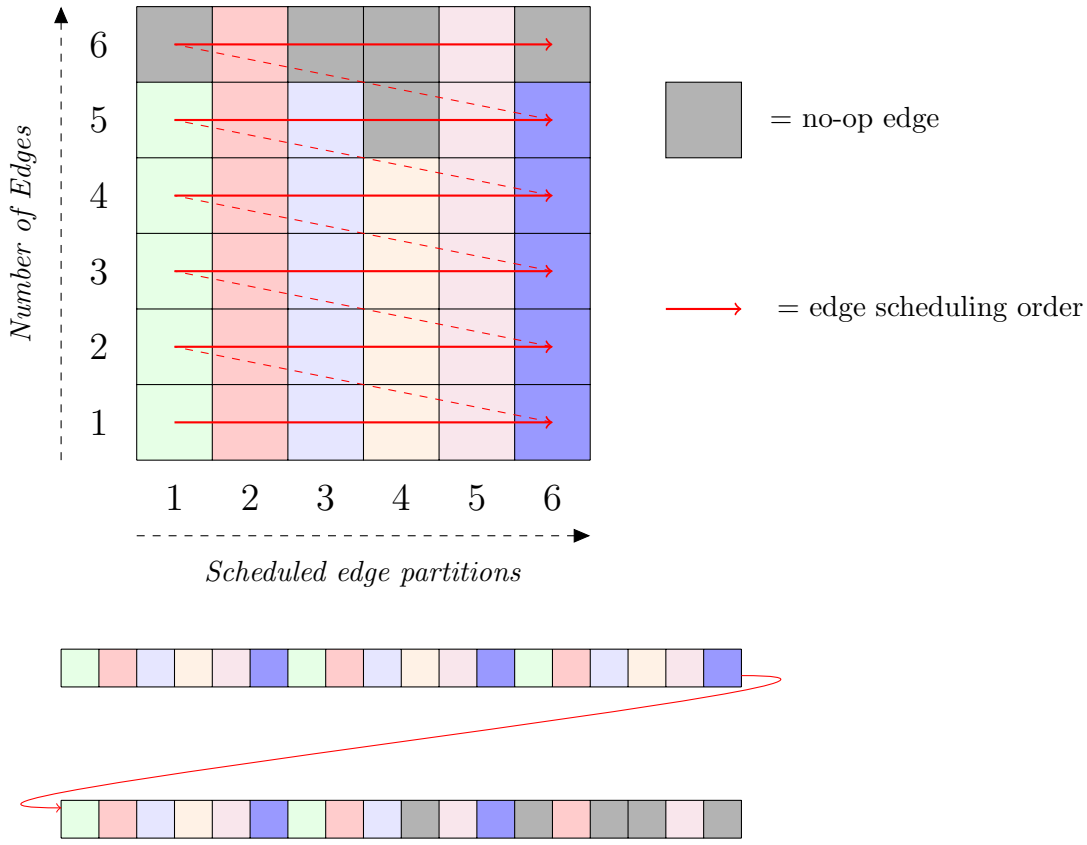


Figure 4.3: Diagram showing the conversion of an edge-partition schedule (top) to an edge schedule (bottom) with the no-op edges shown in gray.

since they take up pipeline stages in order to preserve the independency between each pipeline stage, but producing no useful arithmetic result. In practice, we introduce no-op cells and no-op nodes in the accelerator that when processed (upon encountering a no-op edge) will produce NaN (Not a Number) floating point values that will be ignored by the result committing mechanism. The number of no-op edges depends on how uneven the partitioning is. In practice, the disparity between the number of edges in each edge-partition increases in proportion to the size of the partitions. We can ask METIS to produce as even partitions as possible, but there will always be some disparities.

4.2.2 Complexity

The algorithm described above (Algorithm 3) will find a correct schedule of the adjacency graph, but has a forbiddingly high complexity. Consider the *dual adjacency graph* defined as the nodes of the adjacency graph connected by an edge iff they are not connected in the adjacency graph. The task of scheduling the adjacency graph g with window width l can be transformed into finding a path visiting every node of the dual graph $\overline{g}$ once where each node in the path is adjacent to every node in a window of l around it along that path. Intuitively, we can see that this problem is at least as hard as the problem of finding a Hamiltonian path, which is known to be NP-complete [20] and thus NP-hard. This has serious implications on the performance of our scheduling algorithm, since NP-complete problems do not have any known fast (polynomial-time) algorithms for solving them. In fact, finding an efficient polynomial time algorithm would be tantamount to proving **P**=**NP**, thus revolutionising computer science as we know it. Algorithm 3 performs little better than a blind search through the space of possible permutations of the nodes, hence its complexity is $O(n) = n!$ where n is the number of nodes in the graph. This is quite poor complexity for an algorithm that does not scale well at all. For example, a trial attempt at scheduling a graph with 300 nodes with window width 18 did not terminate even after an overnight run. Clearly we must do better.

4.2.3 No-op edge-partitions

Fortunately, we can work around this issue by introducing *no-op edge-partitions*. No-op edge-partitions contain only no-op edges and are conceptually non-adjacent to any node in the adjacency graph (or, conversely, adjacent to every node in the dual adjacency graph) and can therefore be placed at any position in the schedule, thus padding it out and making it easier to find a schedule. We want to add as few no-op partitions as possible since we are trying to reduce the no-op edge count.

4.2.4 Graph colouring and edge scheduling revisited

A trivial solution for scheduling a graph with window-width l would be to add l no-op partitions after each node, but this would increase the edge count in the mesh by a huge amount, adding $l \times e \times n$ no-op edges, where e is the maximum number of edges per edge-partition and n is the number of nodes in the adjacency graph or the number of edge-partitions. In this section we transform the scheduling problem (Hamiltonian path problem)

to a graph colouring problem for which we can compute a sub-optimal but tolerably good solution in polynomial time. The approach we use utilises graph colouring to group nodes together and using no-op partitions to pad the space between different colours. We define a colouring of a graph as an assignment of a colour to each node such that no two adjacent nodes have the same colour. Graph colouring is a popular problem in computer science and a lot of work has been done in this area. Finding an optimal colouring, one which uses the minimal number of colours is known to be NP-complete [21]. However, we use a greedy colouring algorithm to compute a sub-optimal but adequate colouring that enables us to group independent partitions together (Algorithm 5). Algorithm 5 iterates through every node in the graph and tries to assign it the lowest numbered colour not assigned to its neighbours (that's where the greediness shows). In the worst case each node will be assigned a different colour ($ncolours = nn$), forcing the algorithm to go through all $nn - 1$ colours before finding a valid colour for itself. Thus, in the worst case we iterate over all the nodes, then through all colours (which will be $O(nn)$) and then through all the neighbours of each node (again bounded by $O(nn)$) giving us a worst case complexity of $O(nn^3)$ for a graph with nn nodes. In practice we noticed that the algorithm performs well enough and hardly ever encounters the worst case.

Having coloured the adjacency graph, we can now schedule it with a window width l by grouping the nodes with the same colour one after another, safely knowing that they do not interfere. Transitioning from one colour to the next though can present a problem, since nodes with different colours are not guaranteed to be independent. We mitigate this by adding l no-op edge partitions after each colour group. This means adding $l \times c \times e$ no-op edges to the schedule where c is the number of colours. Evidently, the fewer colours we use the less no-op edges are added, so an optimal colouring of the adjacency graph is desirable, but cannot be computed efficiently.

With this approach we are guaranteed to add no-op edge-partitions. Recall that for a colouring with c colours of a graph that we are trying to schedule with window width l such that the maximum number of edges in each edge partition is e, the number of no-op edges we add is $N_{nop} = l \times c \times e$. We want to minimise N_{nop}. l is a problem parameter and is a characteristic of the hardware. c depends on the quality of the colouring as discussed above. e can be minimised by using smaller edge-partitions. Using smaller edge-partitions means creating more of them per micro-partition. Therefore, using this scheme, it is in our best interests to use as many edge-partitions as possible.

```
function int[ ] colourGraph(Graph g)
    nn ← number of nodes in g
    colours ← int[nn]
    for all node n in g do
        colours[n] ← −1                    ▷ Initialise colours to invalid colour
    end for
    ncolours ← 1
    for all node n in g do
        c ← 0
        repeat ← TRUE
        while repeat do
            available ← TRUE
            for all node neigh adjacent to n and while available do
                if colours[neigh] = c then
                    available ← FALSE
                end if
            end for
            if available then
                colours[n] ← c
                repeat ← FALSE
            end if
            c ← c + 1
            if c = ncolours then
                ncolours ← ncolours + 1
            end if
        end while
    end for
    return colours
end function
```

Algorithm 5: A greedy colouring algorithm that takes an undirected graph g and returns an array containing the colour of each node in g

4.3 Data sets and padding

As dicussed earlier, the kernel on the accelerator has 6 inputs and two outputs:

1. cell data from DRAM
2. node data from DRAM
3. halo cell data from PCIe
4. halo node data from PCIe
5. Edge/address data from DRAM
6. Size vectors from DRAM
7. *res* cell data to DRAM
8. *res* halo cell data to PCIe

The memory system on the MAX3 card places certain limitations on the size of the data streams. In particular: PCIe stream widths must be multiples of 128 bits, while DRAM streams must fit into multiples of 1536 bits. These limitations force us to pad out data sets so that they can be streamed in and out of the accelerator. Remember that cell data consists of the q and *adt* data sets and the *res* result. The node data is just the x data set. The edge data is a vector of the addresses into the BRAMs while the size vectors contain the sizes of the constituent parts of a partition (number of cells, nodes, halo sizes, IPH size etc). For the types of the individual data sets refer to Table 2.1 on page 14. We represent real numbers ($\mathbb{R}$) with 32-bit IEEE-754 floating point numbers. The padding required for each data set is shown in Table 4.1. Notice how in some cases the padding is quite extensive, in the case of node data doubling the size of the input.

The padding will introduce a degradation in performance, since bandwidth is used to transfer these unutilised data items. Due to time limitations we have not had an opportunity to explore more efficient data packing and compression techniques, but it is certainly something worth looking into in future work. For now we are interested in getting a working solution so that we can identify implementation issues such as this.

I/O stream	Original width(bits)	Padding(bits)	Total width(bits)
cell data from DRAM	160	96	256
node data from DRAM	64	64	128
halo cell data from PCIe	160	96	256
halo node data from PCIe	64	64	128
Edge/address data from DRAM	56	8	64
Size vectors from DRAM	238	18	256
res cell data to DRAM	128	0	128
res halo cell data to PCIe	128	0	128

Table 4.1: Table showing the amount of padding added to each dataset to enable streaming.

4.4 FPGA accelerator

4.4.1 I/O streams and manager

The accelerator architecture design has been discussed in the design chapter and its implementation follows those guidelines. A high-level diagram of the input and output streams is shown in Figure 4.4. In practice there is also an extra stream from PCIe directly to DRAM that is used to load the data into the DRAM upon initialisation from the host, but it is ommited from this diagram. The DRAM streams need *address generators* that define the memory access pattern and tell the system where to fetch data from. In our architecture the DRAM address generators provide a simple linear, contiguous stream of increasing addresses for each of the DRAM data streams so that the memory controller can take advantage of the locality and provide the maximum bandwidth possible.

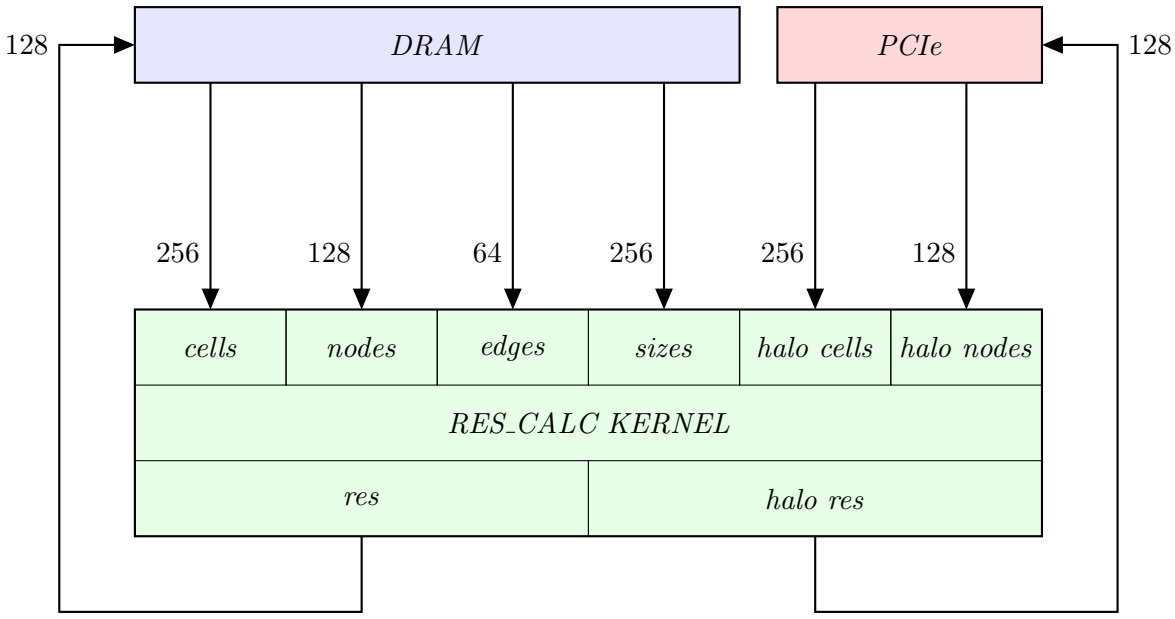


Figure 4.4: The manager graph of the accelerator showing the widths of the I/O streams. The host communicates with the FPGA through the PCIe channel.

4.4.2 Result RAM division and duplication

The kernel itself is programmed using the MaxCompiler API. As discussed earlier, resulting *res* vectors will be stored in local BRAM storage. The RAMs on the Xilinx Virtex6 chip have a limitation of only two ports, meaning that at any single cycle they can accept two address streams, two input data streams, two write-enable signals and produce two output streams. This presents a challenge to the implementation. Recall that, because of the accumulator latency, we must read one value from the result RAM and write the corresponding result of the addition l cycles later, where l is the latency of the accumulator. The only way to implement this is to read the current value of *res* for the cell we are processing from port A and write the corresponding result of the accumulation l cycles later on port B. This is a fine approach, but it presents a difficulty when we want to read out the RAMs for output. Remember that thanks to our two-level partitioning scheme and the resulting interleaving of computation and I/O we are potentially processing and edge and outputting the result of the previous micro-partition, thus needing access to the BRAMs from three different ad-

dresses.

To overcome this, we notice that this contention for addresses happens only during phase 3 of the accelerator where we are processing the second micro-partition and the intra-partition halo while writing out the first micro-partition. We know that the first micro-partition will not be accessed by the edge data (because of the two-level partitioning scheme). We therefore make the decision to split the RAMs for storing the *res* data into two RAMs, one for each micro-partition. The intra-partition halo will be stored in the RAM of the second micro-partition, since it is written out together with the second micro-partition.

This limit on port numbers drives us towards another design decision. Recall that the processing of each edge requires the update of two cells. But we have already used up both ports on the result RAMs for processing just one cell. Therefore we make the decision to use two sets of RAMs, one for each cell access. Of course, this means that the results stored in the RAMs will only be partial *res* vectors that must be summed up on output. Such an arrangement is shown in Figure 4.5. We end up with 8 RAMs for the result data since the arrangement in Figure 4.5 is repeated again for the halo RAMs.

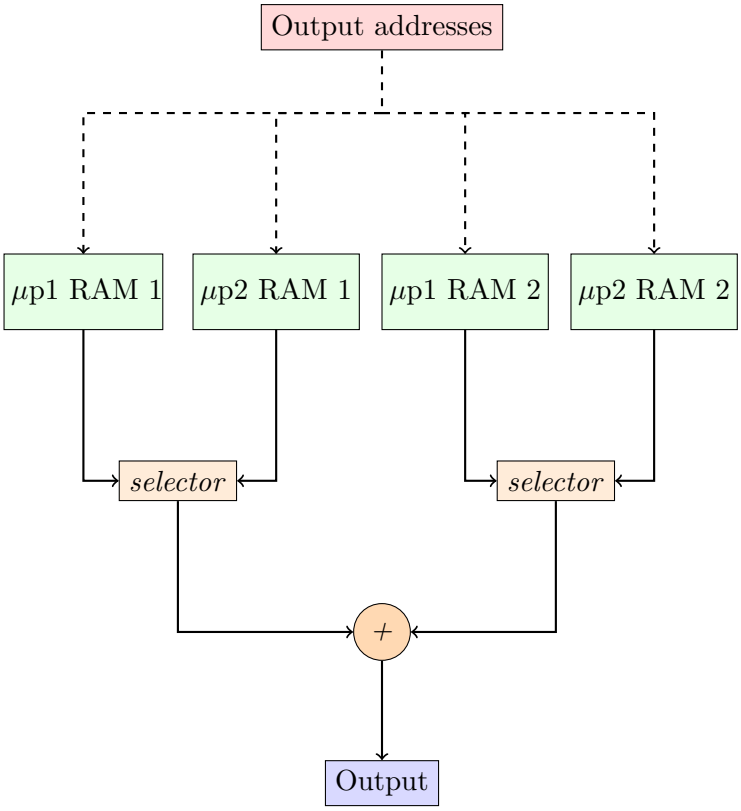


Figure 4.5: Diagram showing the division of the result RAMs and the data flow during the output phase. The selectors choose whether to read from the first or the second micro-partition, dependent on the accelerator phase. The addition node adds the partial results before writing them back. The normal and halo data RAMs are shown here merged, but they are in fact set up as shown in the design chapter.

4.4.3 Resource usage

The FPGA has a limited amount of resources and any extensions to the design must not require more resource than are available. Table 4.2 shows the resource usage of our design. Notice that the arithmetic pipeline uses all of the DSP resources that are used in multiplication operations, but does not consume too much. This gives an indication that adding more pipelines would not be a problem, at least from a resource usage view. MaxCompiler uses extra block RAM resources to schedule the data flow graph and for other low-level "plumbing".

	LUTs	FFs	BlockRAMs	DSPs
Total available	297600	595200	1064	2016
Total used	83863	113985	649	62
Used by kernel	37989	45524	433	62
Used by arithmetic pipeline	17426	22120	0	62
Total used as percentage of available	28.18%	19.15%	60.95%	3.08%

Table 4.2: Table showing the resource usage of our implementation

4.4.4 State machine

The inputs, outputs and enable signals for all elements in the kernel are controlled by a state machine. This is required to implement the overlapping of I/O and execution that we are trying to achieve from the two-level partitioning scheme. The state machine is implemented as a collection of hardware counters that count up to an appropriate maximum defined by the size vector input. For example: in phase 2 (reading in micro-partition 2, processing micro-partition 1) there are counters counting the number of cells and nodes we have read in from DRAM and PCIe and also a counter counting the number of edges we have read in for the processing stage. Once the counters have reached their maximum, that is the number of the relevant elements in the micro-partition, the enable signals for reading and writing are turned off.

Because the bandwidth of halo exchange is slower than that of the DRAM, the halo read/write actions must be delayed since the kernel operates in lock-step, stalling on unavailable inputs. The state machine provides the addresses for writing out the result RAMs. This is a simple matter of writing out the RAMs serially, so we implement this with standard hardware counters. An example of how a counter can be implemented in hardware is shown in Figure 4.6. MaxCompiler provides an API for constructing much more complicated counter configurations with different wrapping protocols, incrementing modes and edge cases. We use counters mostly to generate RAM addresses that are used to store the node and cell data during the

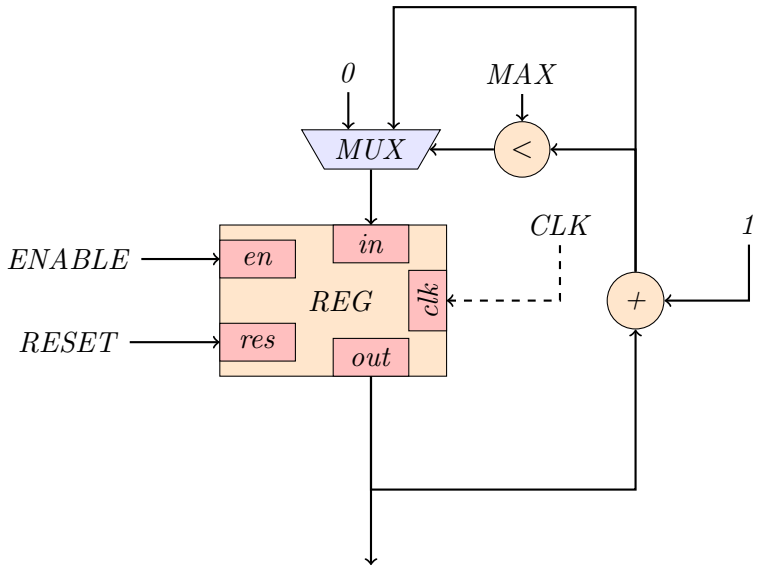


Figure 4.6: Diagram of a hardware counter that increments by one up to a value of *MAX*. A register is used to store the state and has the usual *ENABLE* and *RESET* signals tha can be produced by the kernel state machine. We compare the output with the maximum permitted value and use the result together with a multiplexer to choose whether to wrap around or continue incrementing. The clocking of the design is done automatically by MaxCompiler, so the clock signal is shown here only for completeness.

read in phase and to read the results during the write out phase. The kernel state machine also uses counters to keep track of the I/O and processing state.

Chapter 5

Experimentation

5.1 No-op edges

During our implementation we commented on the necessity of adding no-op edges that arise from the uneveness of the partitioning and the complexity of scheduling. As dicussed in the performance model (section 3.6), for an architecture with less than four pipelines the execution time is dominated by the edge processing. Therefore we want to minimise the number of no-op edges we add. In the section on graph colouring (4.2.4) we predicted that using smaller but more edge-partitions will reduce the number of no-op edges we add due to METIS producing more even partitions and also since the ratio of edge-partitions to no-op edge-partitions will increase. We test this hypothesis by using our test mesh of 721801 nodes, 1438600 edges and 720000 cells and varying the number of edge-partitions per micro-partition and recording the number of no-op edges our graph colouring-based scheduling algorithm ends up adding. The window-width l is 18. The results are shown in Figure 5.1. We notice varying the number of edge-partitions does indeed affect the number of no-op edges added significantly with the number of no-op edges added ranging from 2064540 (ratio 1.44) for 80 edge partitions down to 435088 (ratio 0.3) for 920 edge-partitions. The decrease, however, seems to have unexpected fluctuations that seem to become more and more pronounced as we increase the number of edge-partitions. We believe the cause of these to be inconsistent partitioning from the METIS tool.

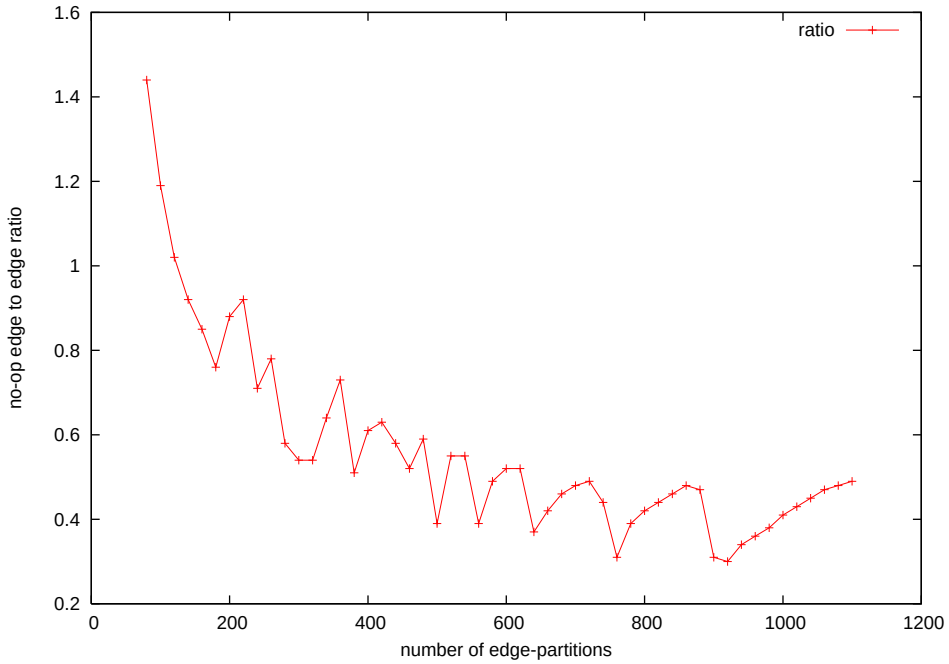


Figure 5.1: Plot of the ratio of no-op edges to normal edges against the number of edge-partitions per micro-partition.

Bibliography

[1] MB Giles, GR Mudalige, Z Sharif, G Markall, PHJ Kelly,
Performance Analysis of the OP2 Framework on Many-core Architectures.
ACM SIGMETRICS Performance Evaluation Review, 38(4):9-15, March 2011

[2] G.R Mudalige, MB Giles, C. Bertolli, P.H.J. Kelly,
Predictive Modeling and Analysis of OP2 on DistributedMemory GPU Clusters
PMBS '11 Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems Pages 3-4

[3] Xilinx Inc.
ISE Design Suite Software Manuals and Help
http://www.xilinx.com/support/documentation/dt_ise11-1.htm

[4] Xilinx Inc.
Virtex-6 Family Overview
http://www.xilinx.com/support/documentation/virtex-6.htm

[5] Maxeler Technologies
MaxCompiler White Paper
http://www.maxeler.com/content/briefings/MaxelerWhitePaperMaxCompiler.pdf

[6] G. Karypis, V. Kumar.
A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs
SIAM Journal on Scientific Computing, Vol. 20, No. 1, pp. 359392, 1999.

[7] IEEE
IEEE Std 754-2008
Publication Year: 2008 , Page(s): 1 - 58

[8] F. B. Kjolstad, M Snir
Ghost Cell Pattern
ParaPLoP '10 Proceedings of the 2010 Workshop on Parallel Programming Patterns

[9] M. T. Jones, K. Ramachandran
Unstructured mesh computations on CCMs
Advances in Engineering Software - Special issue on large-scale analysis, design and intelligent synthesis environments Volume 31 Issue 8-9, Aug-Sept. 2000

[10] H. Morishita, Y. Osana, N. Fujita, H. Amano
Exploiting memory hierarchy for a Computational Fluid Dynamics accelerator on FPGAs
ICECE Technology, 2008. FPT 2008. pp 193 - 200

[11] Sanchez-Roman, D.; Sutter, G.; Lopez-Buedo, S.; Gonzalez, I.; Gomez-Arribas, F.J.; Aracil, J.; Palacios, F.;
High-Level Languages and Floating-Point Arithmetic for FPGABased CFD Simulations
Design & Test of Computers, IEEE, 2011, Volume: 28 Issue:4, pp 28 - 37

[12] Sanchez-Roman, D.; Sutter, G.; Lopez-Buedo, S.; Gonzalez, I.; Gomez-Arribas, F.J.; Aracil, A.;
An Euler Solver Accelerator in FPGA for computational fluid dynamics applications
Proceedings of the 2011 VII Southern Conference on Programmable Logic Crdoba, Argentina April 13 15, 2011

[13] Durbano, J.P.; Ortiz, F.E.;
FPGA-based acceleration of the 3D finite-difference time-domain method
12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004. FCCM 2004.

[14] White B., McKee S. ,de Supinski B., Miller B., Quinlan D., Schulz M., Lawrence Livermore National Laboratory
Improving the computational intensity of unstructured mesh applications
ICS '05 Proceedings of the 19th annual international conference on Supercomputing Pages 341 - 350

[15] Sung-Eui, Y., Lindstrom, P.
Mesh Layouts for Block-Based Caches

IEEE Transactions on visualization and computer graphics, Vol. 12, No. 5, September/October 2006

[16] Shirazi, N., Walters, A., Athanas, P.
Quantitative analysis of floating point arithmetic on FPGA based custom computing machines
IEEE Symposium on FPGAs for Custom Computing Machines, 1995. Proceedings.

[17] Hartstein, A. Puzak, Thomas R.
The Optimum Pipeline Depth for a Microprocessor
ISCA '02 Proceedings of the 29th annual international symposium on Computer architecture Pages 7 - 13.

[18] Khronos Group
The OpenCL Specification v1.2
www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[19] NVIDIA
Tesla M2050 / M2070 GPU Module Specification Document
http://www.nvidia.com/docs/IO/43395/BD-05238-001_v03.pdf

[20] Trummel, K. E. ; Weisinger, J. R.
The Complexity of the Optimal Searcher Path Problem
Operations Research , Vol. 34, No. 2 (Mar. - Apr., 1986), pp. 324-327

[21] Karp, Richard M.
50 Years of Integer Programming 1958-2008
Springer Berlin Heidelberg, ISBN:978-3-540-68279-0, pp.219-241

Appendix A

Code Samples

A.1 Airfoil Kernel definitions in C

Even though we focused on accelerating the res_calc kernel, the other kernels are shown here for the sake of completeness.

```
void adt_calc(float *x1,float *x2,float *x3,float *x4,float *q,float *adt){
  float dx,dy, ri,u,v,c;
  ri = 1.0f/q[0];
  u = ri*q[1];
  v = ri*q[2];
  c = sqrt(gam*gm1*(ri*q[3]-0.5f*(u*u+v*v)));
  dx = x2[0] - x1[0];
  dy = x2[1] - x1[1];
  *adt = fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
  dx = x3[0] - x2[0];
  dy = x3[1] - x2[1];
  *adt += fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
  dx = x4[0] - x3[0];
  dy = x4[1] - x3[1];
  *adt += fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
  dx = x1[0] - x4[0];
  dy = x1[1] - x4[1];
  *adt += fabs(u*dy-v*dx) + c*sqrt(dx*dx+dy*dy);
  *adt = (*adt) / cfl;
}
```

```
void bres_calc(float *x1, float *x2, float *q1,
                      float *adt1,float *res1, int *bound) {
  float dx,dy,mu, ri, p1,vol1, p2,vol2, f;
  dx = x1[0] - x2[0];
  dy = x1[1] - x2[1];
  ri = 1.0f/q1[0];
  p1 = gm1*(q1[3]-0.5f*ri*(q1[1]*q1[1]+q1[2]*q1[2]));
  if (*bound==1) {
    res1[1] += + p1*dy;
    res1[2] += - p1*dx;
  }
  else {
    vol1 = ri*(q1[1]*dy - q1[2]*dx);

    ri = 1.0f/qinf[0];
    p2 = gm1*(qinf[3]-0.5f*ri*(qinf[1]*qinf[1]+qinf[2]*qinf[2]));
    vol2 = ri*(qinf[1]*dy - qinf[2]*dx);

    mu = (*adt1)*eps;

    f = 0.5f*(vol1* q1[0] + vol2* qinf[0] ) + mu*(q1[0]-qinf[0]);
    res1[0] += f;
    f = 0.5f*(vol1* q1[1] + p1*dy + vol2* qinf[1] + p2*dy) + mu*(q1[1]-qinf[1]);
    res1[1] += f;
    f = 0.5f*(vol1* q1[2] - p1*dx + vol2* qinf[2] - p2*dx) + mu*(q1[2]-qinf[2]);
    res1[2] += f;
    f = 0.5f*(vol1*(q1[3]+p1) + vol2*(qinf[3]+p2) ) + mu*(q1[3]-qinf[3]);
    res1[3] += f;
  }
}

void res_calc(float *x1, float *x2, float *q1, float *q2,
                     float *adt1,float *adt2,float *res1,float *res2) {

  float dx,dy,mu, ri, p1,vol1, p2,vol2, f;

  dx = x1[0] - x2[0];
  dy = x1[1] - x2[1];

  ri = 1.0f/q1[0];
```

```
  p1 = gm1*(q1[3]-0.5f*ri*(q1[1]*q1[1]+q1[2]*q1[2]));
  vol1 = ri*(q1[1]*dy - q1[2]*dx);

  ri = 1.0f/q2[0];
  p2 = gm1*(q2[3]-0.5f*ri*(q2[1]*q2[1]+q2[2]*q2[2]));
  vol2 = ri*(q2[1]*dy - q2[2]*dx);

  mu = 0.5f*((*adt1)+(*adt2))*eps;

  f = 0.5f*(vol1* q1[0] + vol2* q2[0] ) + mu*(q1[0]-q2[0]);
  res1[0] += f;
  res2[0] -= f;
  f = 0.5f*(vol1* q1[1] + p1*dy + vol2* q2[1] + p2*dy) + mu*(q1[1]-q2[1]);
  res1[1] += f;
  res2[1] -= f;
  f = 0.5f*(vol1* q1[2] - p1*dx + vol2* q2[2] - p2*dx) + mu*(q1[2]-q2[2]);
  res1[2] += f;
  res2[2] -= f;
  f = 0.5f*(vol1*(q1[3]+p1) + vol2*(q2[3]+p2) ) + mu*(q1[3]-q2[3]);
  res1[3] += f;
  res2[3] -= f;
}

void save_soln(float *q, float *qold){
  for (int n=0; n<4; n++) qold[n] = q[n];
}

void update(float *qold, float *q, float *res, float *adt, float *rms){
  float del, adti;

  adti = 1.0f/(*adt);

  for (int n=0; n<4; n++) {
    del = adti*res[n];
    q[n] = qold[n] - del;
    res[n] = 0.0f;
    *rms += del*del;
  }
}
```

List of Figures